Apport de la méta-modélisation formelle pour la conception des Systèmes Automatisés de Production

Laurent Piétrac

DOCTORAT DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Spécialité : **AUTOMATIQUE**

THÈSE

présentée par

Laurent PIÉTRAC

Pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Apport de la méta-modélisation formelle pour la conception des Systèmes Automatisés de Production

Soutenue le 12 janvier 1999 devant le jury composé de :

Jean-Louis Ferrier	président du jury
Henri Habrias	rapporteur
François Vernadat	rapporteur
Pierre Bourdet	examinateur
Bruno Denis	examinateur
Éric Niel	examinateur
Jean-Jacques Lesage	directeur de thèse

Laboratoire Universitaire de Recherche en Production Automatisée
École Normale Supérieure de Cachan
61, avenue du Président Wilson 94235 CACHAN CEDEX

Avant-propos

Le travail présenté dans ce mémoire a été effectué, au sein du Laboratoire Universitaire de Recherche en Production Automatisée (LURPA - EA 1385) de L'École Normale Supérieure de Cachan, sous la direction de M. le professeur Jean-Jacques Lesage.

La présidence du jury de cette thèse a été assurée par M. le professeur Jean-Louis Ferrier. Il a su rendre l'ambiance de cette soutenance très agréable, qu'il en soit grandement remercié.

Les rapports auprès de la formation doctorale ont été établis par M. le professeur Henri Habrias et M. le professeur François Vernadat. Je les remercie vivement d'avoir accepté cette tâche, ainsi que pour leurs nombreuses questions constructives.

Je tiens à exprimer toute ma reconnaissance au professeur Pierre Bourdet pour m'avoir accueilli dans le laboratoire qu'il dirige, et pour avoir participé au jury de cette thèse.

M. le maître de conférence Bruno Denis a co-encadré ce travail de thèse. Je tiens à le remercier pour son esprit d'analyse, sa disponibilité et sa sympathie. Je tiens aussi à le remercier pour avoir su me laisser un maximum de liberté dans mon travail.

Je tiens à remercier également M. le maître de conférence Éric Niel, habilité à diriger des recherches, pour avoir participé à ce jury de thèse. J'espère que cette soutenance aura été le début d'une longue collaboration.

M. le professeur Jean-Jacques Lesage a co-encadré ce travail de thèse. Je tiens à le remercier pour m'avoir accueilli au sein de son équipe et de m'avoir fait confiance tout au long de ces années. Son esprit d'analyse, son recul par rapport aux travaux effectués, sa diplomatie sont pour moi des exemples.

A propos d'exemple, je tiens à remercier Emmanuel Duc, avec qui j'ai partagé un aquarium pendant ces années de thèse. Maintenant que je suis solitaire dans un nouveau bureau, ses chants mélodieux me manquent presque ... Ce qui me manquera par contre c'est son amitié sans faille et nos longues discussions. Si nous ne referons pas le monde, je suis certain que lui fera avancer la recherche.

Je tiens enfin à remercier tous les membres du LURPA avec qui j'ai passé de très agréables moments. Merci également à tous les membres de ma famille, et à mes amis, qui ont également porté le poids de ces années de travail intense.

Table des matières

Table des figures

Introduction

Les Systèmes Automatisés de Production (SAP) sont des systèmes complexes dont la conception fait appel à de nombreux métiers. Les acteurs de cette conception s'intéressent à des parties ou à des aspects différents du système, interviennent à des étapes différentes de la conception. Dans les premières phases de la conception, avant sa réalisation effective, le système n'est vu qu'au travers de modèles. Ces modèles sont à la fois le résultat de leur travail et le support de communication qui constitue le lien entre ces différents spécialistes. Les modèles sont donc d'une importance cruciale lors de la conception d'un SAP : un SAP de qualité ne peut être conçu sans modèles de qualité.

Les travaux portant sur l'amélioration de la qualité des modèles pour les SAP ont exploré deux voies principales. La première a pour objectif de mieux définir le rôle de chaque activité et ainsi d'assurer la cohérence des activités les unes par rapport aux autres. Quant à la seconde, elle s'intéresse plutôt à la rigueur des modèles construits et à l'intégration de ces différents modèles.

Pour assurer la cohérence entre les différentes étapes, certains travaux proposent un cycle de modélisation dans lequel chacune des activités est située par rapport aux autres. Le résultat attendu de chaque activité est précisé et finalement seul le choix du langage et de la méthode utilisés sont laissés aux concepteurs [Chlique92]. Une approche moins restrictive consiste à définir un cadre sémantique utilisé pour distinguer et intégrer les différentes étapes de modélisation [Vernadat93]. Le choix du langage et de la méthode est toujours libre, mais, en plus, la succession des activités n'est pas figée.

Les travaux étudiant la rigueur et l'intégration des modèles utilisent la méta-modélisation [Bon bierel98]. Modéliser consiste à construire un modèle d'un SAP. Méta-modéliser consiste à construire un modèle d'un modèle d'un SAP. Par extension, un méta-modèle peut aussi modéliser une méthode de conception d'un SAP. Dans la plupart des cas, les méta-modèles n'étudient que la structure d'un modèle. Les aspects construction et comportement dynamique ne sont souvent pas étudiés. Lorsqu'ils le sont, ces études sont toujours complètement distinctes de la méta-modélisation des données. De plus, les langages utilisés dans ces travaux ne permettent pas de valider et de vérifier les méta-modèles.

Nous pensons également que pour avoir une chance de concevoir des SAP de qualité,

fiables, adaptatifs et répondant aux besoins exprimés, il est indispensable de construire des modèles ayant une syntaxe et une sémantique sans aucune ambiguïté. Nous pensons, par contre, qu'un modèle est un système à part entière et donc qu'une approche cartésienne de son étude, en le découpant en parties « indépendantes », n'est pas cohérente et ne garantit pas la qualité du résultat. Notre objectif, à travers ces travaux, a donc été de proposer une approche de méta-modélisation abordant tous les aspects de la modélisation, c'est-à-dire aussi bien les définitions rigoureuses des langages utilisés, les aspects statiques et dynamiques des modèles, et les méthodes d'utilisation ou de construction de ces modèles. Tout ceci avec un seul langage pour intégrer au maximum ces différents aspects de la modélisation.

Ce langage se doit d'être lui-même le plus rigoureux possible afin que les méta-modèles construits soient sans ambiguïté. Nous avons donc choisi d'utiliser un langage construit sur des théories mathématiques : le langage Z. Basé sur une théorie des ensembles et la logique des prédicats, ce langage récent est déjà utilisé pour la spécification de logiciels. Les concepts utilisés sont très généraux, ce qui rend l'utilisation de Z possible dans des domaines très différents. Notre application en est un exemple concret.

Le premier chapitre de ce mémoire est une présentation des travaux portant sur l'analyse des étapes de modélisation des systèmes automatisés de production. Ce chapitre commence bien sûr par une présentation des SAP, puisqu'ils constituent *in fine* l'objectif de nos travaux. Nous avons pourtant voulu cette présentation courte, car, ce qui nous intéresse, c'est la vision qu'ont les concepteurs de ces SAP, plus que les SAP eux-mêmes. L'objectif est donc de faire un état de l'art des différentes études sur l'activité de modélisation et, en même temps, de présenter l'ensemble du vocabulaire utilisé dans ces études.

Cette étude du vocabulaire nous permet de montrer que l'activité de modélisation n'est pas quelque chose de clairement défini, et que les notions utilisées sont floues ou ambiguës. Le deuxième chapitre est une analyse de cette activité permettant, au fur et à mesure, de construire un ensemble cohérent de définitions des concepts sous-jacents. Ces définitions nous amènent ensuite à présenter les différents aspects de leur étude, et donc à classer l'ensemble des points qui doivent être abordés en méta-modélisation. Ce chapitre constitue notre première contribution à l'étude de la méta-modélisation.

Dans le chapitre trois, nous développons notre deuxième contribution. Elle correspond à l'utilisation d'un langage formel, le langage Z, pour la méta-modélisation des langages et méthodes de conception des SAP. Après une courte justification de l'utilisation de Z plutôt qu'un autre langage formel, nous présentons les différents aspects de Z qui rendent ce langage apte à l'étude complète de l'activité de méta-modélisation. Nous reprenons ensuite chacun des besoins énumérés au chapitre deux pour montrer les détails de la

méta-modélisation avec Z.

Afin de valider notre approche, nous présentons, dans le chapitre quatre, une application sur un langage couramment utilisé pour la modélisation des SAP. Nous avons choisi une classe de réseaux de Petri (RdP), en retenant une définition mondialement connue et utilisée : celle du professeur Murata. Afin de valider le taux de couverture très important de Z, nous présentons aussi bien la définition du langage que la construction d'un modèle ou le joueur d'un tel RdP, ainsi que la vérification et la validation du méta-modèle.

Le chapitre cinq est consacré à la présentation d'une méthode comprenant plusieurs langages. Pour montrer tout l'intérêt du langage Z, nous avons sélectionné une méthode dans laquelle l'intégration ne peut pas être vue sous le seul angle des données, mais doit l'être aussi sous l'aspect dynamique (ou joueur) : aucune autre approche existante de méta-modélisation ne permet cette étude. Cette méthode, issue de travaux du LAAS, intègre à la fois des RdP et des équations différentielles. Cet exemple nous permet, en outre, d'étendre notre méthodologie à la modélisation des systèmes hybrides de production.

Chapitre 1

La conception des SAP

Nos travaux portent sur la conception des systèmes automatisés de production (SAP) et plus particulièrement sur la conception de la commande des systèmes à événements discrets (SED). Ces systèmes sont des systèmes complexes dont les contextes économiques et techniques évoluent rapidement [Kiefer et al.95]. La conception de ces systèmes complexes nécessite un cadre de travail rigoureux intégrant des équipes importantes avec des métiers différents. L'évolution rapide des SAP pousse leurs concepteurs à rechercher sans arrêt des approches de conception différentes pour répondre à de nouveaux besoins.

Dans ce chapitre, nous présentons les travaux apportant une réponse à ces besoins de définition, d'intégration et d'évolution des activités de conception des SAP. Notre objectif n'est pas l'étude des SAP, mais l'étude rigoureuse et complète des activités de conception des SAP. Cependant, ce chapitre commencera par une présentation rapide de la diversité des moyens utilisés. Nous détaillerons ensuite les approches permettant d'organiser les différentes activités de modélisation. La partie la plus importante de ce chapitre sera consacrée aux différentes techniques de méta-modélisation pour la définition de langages ou de méthodes multi-langages.

Tout au long de ce chapitre, de nombreuses approches de la modélisation, faisant chacune appel à un vocabulaire spécifique, vont être évoquées. Pour ne pas dénaturer la pensée de leurs auteurs, nous utiliserons ce vocabulaire spécifique. Pour bien montrer la diversité des termes utilisés, ce vocabulaire apparaîtra **en caractères gras** dans ce document.

1.1 Génie automatique = génie informatique ?

Le système d'information, et le système informatique en général, occupe une part importante du SAP. Il est donc naturel de se demander si les méthodes utilisées pour le Génie Informatique peuvent être utilisées pour le Génie Automatique. De nombreux

travaux, précurseurs dans notre domaine, ont donc consisté, et consistent toujours, à essayer d'adapter des méthodes de l'informatique à la conception des systèmes d'information des SAP. SADT [iT89], Merise [Tardieu et al.83], entité/relation [Chen76], AOO [Coad et al.92] ... sont quelques exemples parmi les plus connus de cette approche.

Une méthode est conçue pour être utilisée pour modéliser certains types de systèmes. Lorsqu'elle est utilisée pour d'autres systèmes, elle peut ne pas correspondre tout à fait à l'ensemble des besoins. Il est alors bien pratique de pouvoir la modifier légèrement pour qu'elle remplisse complètement la tâche qui lui a été assignée. Ainsi, IDEF0 a été utilisé dans [Denis et al.92] pour spécifier le cahier des charges d'une station de dessalement d'eau de mer. Pour rendre les modèles plus lisibles, les flux de données ont été représentés par des traits fins pour les flux d'informations et par des traits gras pour les flux de produits. Cette modification a amélioré la lisibilité des modèles en faisant ressortir, parmi les nombreux flux, ceux du produit traité. Un tel exemple montre bien que, même pour un domaine commun au Génie Automatique et au Génie Informatique (l'étude fonctionnelle dans ce cas), les méthodes du Génie Automatique ne sont pas forcément parfaitement adaptées à l'étude des SAP.

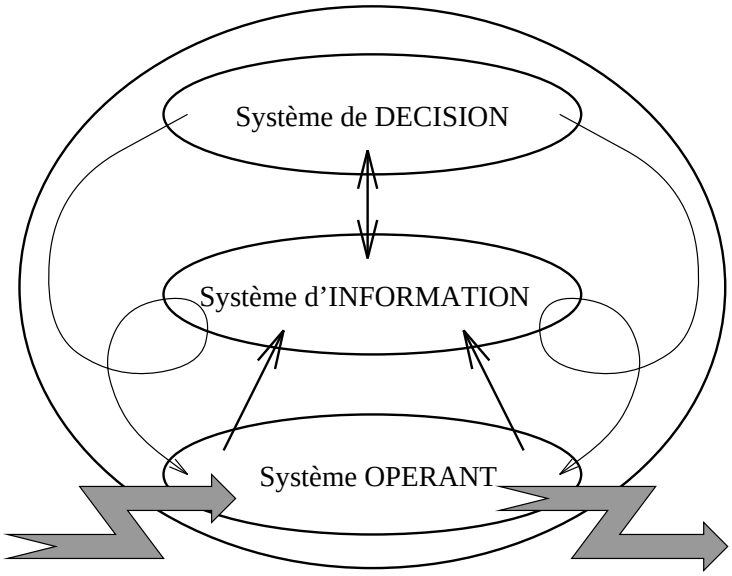


FIG. 1.1: *Modèle canonique O.I.D. [Lemoigne90]*

En outre un SAP n'est pas constitué que d'un système d'information. Un SAP, d'après le modèle canonique O.I.D (figure 1.1) de Jean-Louis Le Moigne [Lemoigne90], peut être décomposé en trois sous-systèmes : le système Opérant, le système d'Information et le système de Décision. Certes, le Génie informatique a beaucoup apporté pour la conception du système d'information des SAP. Par contre, les systèmes opérant et décisionnel ont des besoins propres dont certains aspects sont trop particuliers pour que les méthodes issues

du Génie informatique soient directement utilisables. Des méthodes particulières ont donc été créées pour l'étude des SAP, nous allons maintenant les aborder.

1.2 Les SAP sont des systèmes complexes !

1.2.1 Des besoins particuliers

Le Grafcet et le GEMMA sont deux exemples **d'outils de modélisation** créés spécifiquement pour le Génie Automatique. Le « Guide d'étude des modes de marches et d'arrêts » (GEMMA) a été élaboré en 1981 [Adepa81]. Le « Graphe de Commande Étape/Transition » (Grafcet) a été normalisé pour la première fois en 1982 [Afnor82]. La création de ces deux outils a été une réponse brillante à un manque pour la conception du système décisionnel des SAP. Cependant, depuis quinze ans les systèmes automatisés de production ont beaucoup évolué, et cette évolution pose parfois des problèmes dans l'utilisation de tels outils.

Ainsi, le GEMMA a été créé pour l'étude d'un système comportant une partie opérative et une partie commande. Pour un tel système, il aide le concepteur dans l'étude des différents modes de marche possibles du système et des évolutions possibles entre modes de marche. Par sa lecture aisée, la **grille** GEMMA facilite le travail de conception tout en permettant une étude exhaustive. Aujourd'hui les SAP sont constitués de sous-systèmes, répartis, coopérants ... qui ont tous différents modes de marche. Malheureusement le GEMMA n'a pas été prévu pour ce genre de systèmes. Bien sûr, l'utilisation du GEMMA reste possible, mais sous réserve d'hypothèses locales importantes [Apave92], [Lesage et al.93]. Le manque d'évolution du GEMMA pénalise donc son utilisation face à des SAP qui ont évolués.

Le **diagramme fonctionnel** Grafcet par contre n'a pas cessé d'évoluer. D'abord normalisé en France en 1982 [Afnor82] puis en 1993 [Afnor93], il l'a été au niveau international en 1988 [Cei88]. Pour le Grafcet, au contraire du GEMMA, c'est plutôt son évolution qui pose problème. En effet le diagramme fonctionnel est défini sous forme textuelle, en français ou en anglais. Les **éléments, règles et structures de base** sont présentés et expliqués à l'aide d'exemples. La seule utilisation du texte comme référence ne permet malheureusement pas d'assurer la cohérence et l'unicité d'interprétation d'un document. Le problème est encore plus important lorsqu'il s'agit d'assurer la cohérence de plusieurs normes qui traitent du même sujet.

Les remarques faites ici ne sont pas seulement valables pour le Grafcet ou le GEMMA qui sont des outils plutôt bien définis. Notre propos n'était pas ici de critiquer ces outils mais plutôt de regretter que les méthodes employées pour les définir ne permettent pas

d'assurer la pérennité de ces outils. Pour pouvoir durer, et encore répondre aux besoins, les outils du Génie Automatique doivent avoir des définitions rigoureuses, structurées et ouvertes à leurs utilisateurs : une définition textuelle ne peut pas répondre à ces besoins.

1.2.2 Des méthodes plus globales

Quelques travaux consistent à associer plusieurs approches au sein d'une méthode permettant ainsi :

- soit une étude plus détaillée ou plus facile ou plus rigoureuse d'un aspect du SAP,
- soit une étude (en parallèle ou successivement) de plusieurs aspects du SAP à concevoir.

Citons pêle-mêle quelques unes de ces méthodes :

- l'utilisation conjointe des **langages** Statecharts et Grafcet [Sartor95] pour la spécification des modes de marche ;
- l'utilisation conjointe d'une **approche** orientée objet, avec des **règles** similaires à celles de HOOD, et des Statecharts [Guegen et al.92] [Guegen et al.93] pour la rédaction du cahier des charges de systèmes de contrôle/commande ;
- le passage d'un **modèle** SADT à un **modèle** SADT temporel [Zaytoon93] [Zaytoon et al.93] pour intégrer les **contraintes** de synchronisation et de sécurité à l'approche fonctionnelle ;
- la description fonctionnelle d'un système avec la **méthode** IDEF0, puis la description du comportement des activités élémentaires IDEF0 par approche synchrone (Grafcet pour les activités séquentielles, SIGNAL pour les activités continues) [Gerval et al.92].

Toutes ces approches ont, à notre avis, deux inconvénients majeurs. D'une part, les différentes approches utilisées n'ont pas été réétudiés par les auteurs qui se sont plutôt attachés à l'étude de leur intégration. Si ces approches ne sont pas définies de manière suffisamment rigoureuse, la méthode globale manquera également de rigueur. De plus, ces méthodes manquent de pérennité et de flexibilité. De pérennité, parce que les approches elles-mêmes manquent de pérennité, remarque que nous avons déjà effectuée. De flexibilité car ces méthodes sont prévues pour être utilisées telles qu'elles, de manière rigide, sans pouvoir être adaptées au savoir faire des utilisateurs.

Ces remarques sont d'autant plus importantes lorsque la méthode proposée couvre un éventail très large des besoins en conception des SAP : c'est le cas par exemple de la

méthode GIM [Zanettin94]. Parce qu'elle impose l'utilisation de **méthodes** particulières complémentaires, elle a l'énorme avantage de former un tout cohérent, qui répond de façon structurée à de nombreux besoins du concepteur. A cause de ce choix, elle a l'inconvénient de ne pas pouvoir être adaptée aux habitudes et aux méthodes connues des industriels ce que permettent au contraire les approches plus générales présentées dans la section suivante.

1.3 L'évolution du Génie Automatique

1.3.1 Prendre du recul ...

La carte des activités

A partir du cycle en V de l'AFCIQ de production de logiciels, et du partage d'un SAP en partie commande et partie opérative, le Centre Coopérant en Génie Automatique (CCGA) a défini une « **carte des activités** de développement d'un système automatisé de production » (figure 1.2). Cette carte a pour objectif de préciser les différentes activités nécessaires à la conception complète d'un SAP, bien qu'elle insiste plus sur la partie commande que sur la partie opérative. A chacune des activités, les résultats attendus sont précisés, sans pour autant imposer de démarche de travail. Ce modèle est donc indépendant des moyens permettant d'obtenir ces résultats intermédiaires.

Un avantage important de cette méthode est qu'elle ne remet pas en cause les habitudes des équipes de conception, car elle ne leur impose pas l'utilisation de nouveaux **outils**. En effet elle permet l'utilisation d'outils du marché déjà existants [Chlique92] et déjà utilisés au sein des équipes. Elle peut ainsi être utilisée pour tous les SAP, présents ou futurs.

La carte des activités ne résout pas le problème de cohérence entre les différentes activités. Considérons le cas général où, pour chaque activité, une équipe utilise un ou plusieurs outils. Le point de départ de cette équipe est principalement constitué des modèles produits dans les activités amont. Cette équipe doit donc être capable d'interpréter ces modèles. Il apparaît donc clairement qu'un des problèmes majeurs de la conception du SAP sera un problème de dialogue entre les différentes équipes.

De plus, le cycle en V distingue les opérations de conception de celles de validation. Or les coûts de conception des SAP sont tels qu'il n'est pas concevable d'attendre les dernières étapes pour valider les SAP. Dès les premières étapes de conception, les modèles doivent donc être vérifiés et validés [Zaytoon et al.97] pour permettre une conception de SAP de qualité. Or les différents outils ne sont vus que comme des moyens de produire des modèles qui serviront de base pour les étapes suivantes : leur rigueur n'est pas abordée. Il n'est pourtant pas possible d'obtenir des modèles de qualité sans outils de qualité !

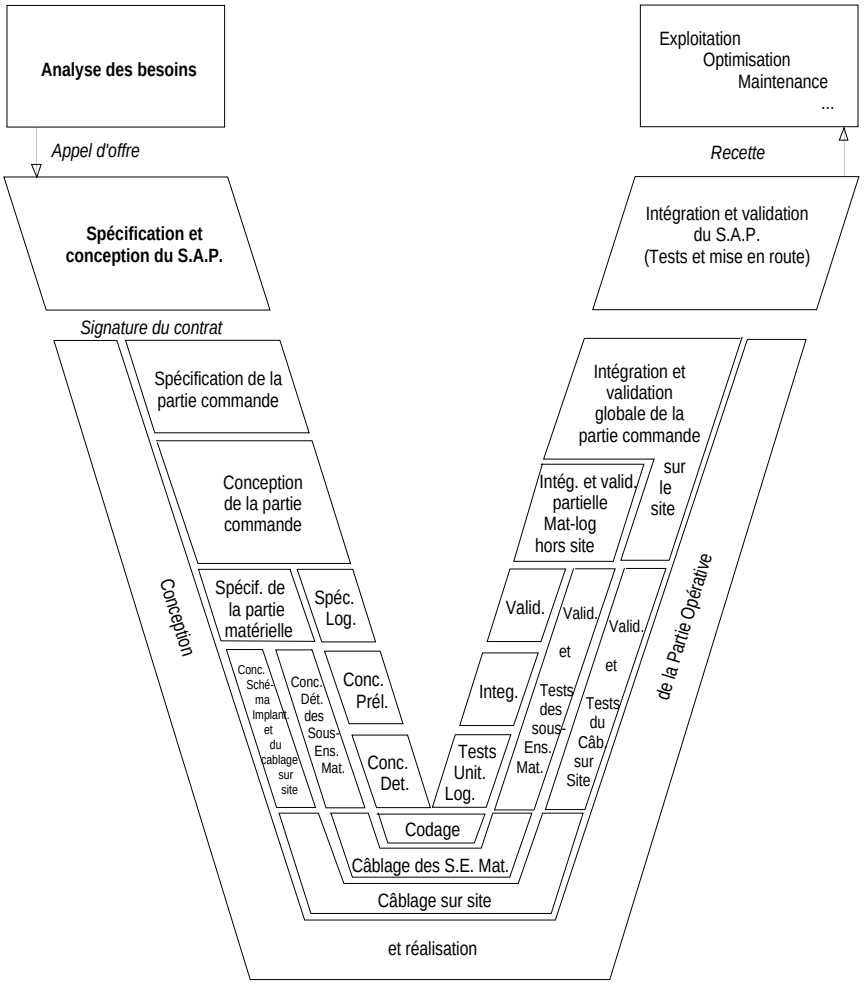


FIG. 1.2: *Carte des activités de développement d'un système automatisé de production*

Les cadres de modélisation

Face à la nécessité de modèles portant sur des besoins différents, CIM-OSA [Consortium89] propose un **cadre de modélisation** (figure 1.3). Ce cadre de modélisation propose une classification suivant trois axes des **modèles** à construire : l'axe de génération, l'axe de dérivation et l'axe de particularisation. En se référant à ce cube, les concepteurs peuvent vérifier que les modèles construits couvrent des aspects différents et complémentaires.

L'axe de génération permet de vérifier que les vues fonctionnelle, information, ressource et organisation du système à concevoir ont toutes été abordées. L'axe de dérivation incite à comparer les étapes de la méthode de conception aux niveaux d'abstraction donnés : les besoins, la conception et l'implantation. Quant à l'axe de particularisation, il oblige à se souvenir que le système à modéliser n'est pas le premier système du domaine, et que la conception peut se faire par particularisation de modèles génériques pour les systèmes de

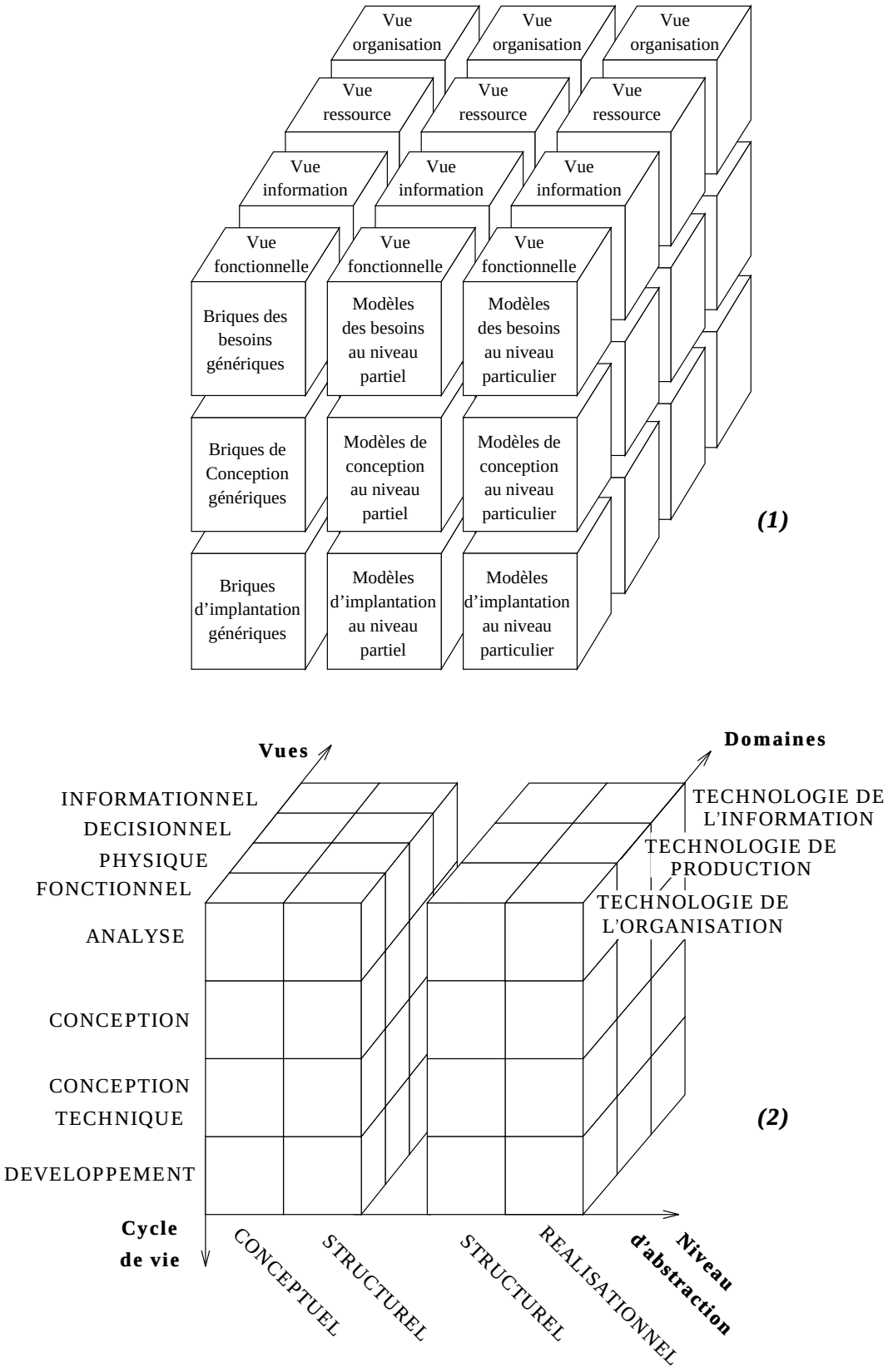


FIG. 1.3: *Les cadres de modélisation de CIM-OSA (1) [Consortium89] et d'IMPACS (2) [Zanettin94]*

production. Le cadre de modélisation CIM-OSA ne propose donc pas une démarche de travail mais il pousse à vérifier une certaine cohérence de la méthode choisie par rapport aux références que sont ses axes.

Le **cadre de modélisation** d'IMPACS [Zanettin94] a un objectif différent sur un point. **L'architecture** IMPACS (figure 1.3) est une architecture générique qui doit être particularisée pour le système à concevoir. L'axe de particularisation n'existe donc pas. Par contre, ce cadre comporte un axe cycle de vie avec les niveaux analyse, conception, conception technique et développement. La coexistence des axes cycle de vie et niveau d'abstraction (conceptuel, structurel, réalisationnel) permet de bien faire la distinction entre les étapes de la méthode et les étapes de réflexion sur le système : cette distinction n'existe pas dans CIM-OSA.

Ces deux cadres de modélisation, CIM-OSA et IMPACS, permettent donc une couverture la plus large possible des différents modèles constructibles pour la conception d'un SAP. Ils n'imposent pas de méthode de travail mais permettent de situer les méthodes par rapport à des besoins. CIM-OSA aussi bien qu'IMPACS proposent des **modèles** associés à chaque case du cube, cependant chaque utilisateur est libre de se servir de ses propres modèles. La cohérence interne à chaque modèle, ainsi que les liens entre eux ne sont donc pas abordés. Encore une fois la difficulté majeure de la conception d'un SAP va se situer dans les phases de communication. Ces cadres n'aident pas le concepteur pour la création de modèles. Ils ne permettent pas de vérifier que les modèles produits sont corrects.

1.3.2 Pour mieux modéliser

Les cadres de travail et des modèles permettent de prendre du recul sur les besoins de la conception des SAP, sans pour autant aider à la modélisation. Afin de mieux définir les modèles utilisés et les méthodes intégrant ces modèles, une nouvelle approche est apparue : la modélisation de ces modèles et méthodes. Ces « modèles de modèles » sont souvent appelés **méta-modèles**. Cette nouvelle approche est, à notre avis, la seule permettant de rendre plus rigoureux mais aussi plus évolutifs ces modèles et méthodes. D'une part la construction de méta-modèles oblige à structurer sa vision du modèle ou de la méthode modélisée. D'autre part, les méta-modèles peuvent être modifiés en fonction des habitudes ou de nouveaux besoins des utilisateurs, tout en obligeant le méta-modélisateur à réfléchir à la conservation de la cohérence du méta-modèle produit.

Cette conviction explique que nous consacrions toute la section suivante à l'étude des différents travaux existants sur la méta-modélisation de modèles ou méthodes employés pour l'étude des SAP. Nous commencerons par une approche pragmatique, GRAPHTALK, qui fut certainement la première approche industrielle de méta-modélisation. Nous présenterons ensuite les approches entité/relation, traitements et construction de modèles

avant de présenter les travaux sur l'intégration des modèles.

1.4 Les travaux de méta-modélisation

1.4.1 Une approche industrielle

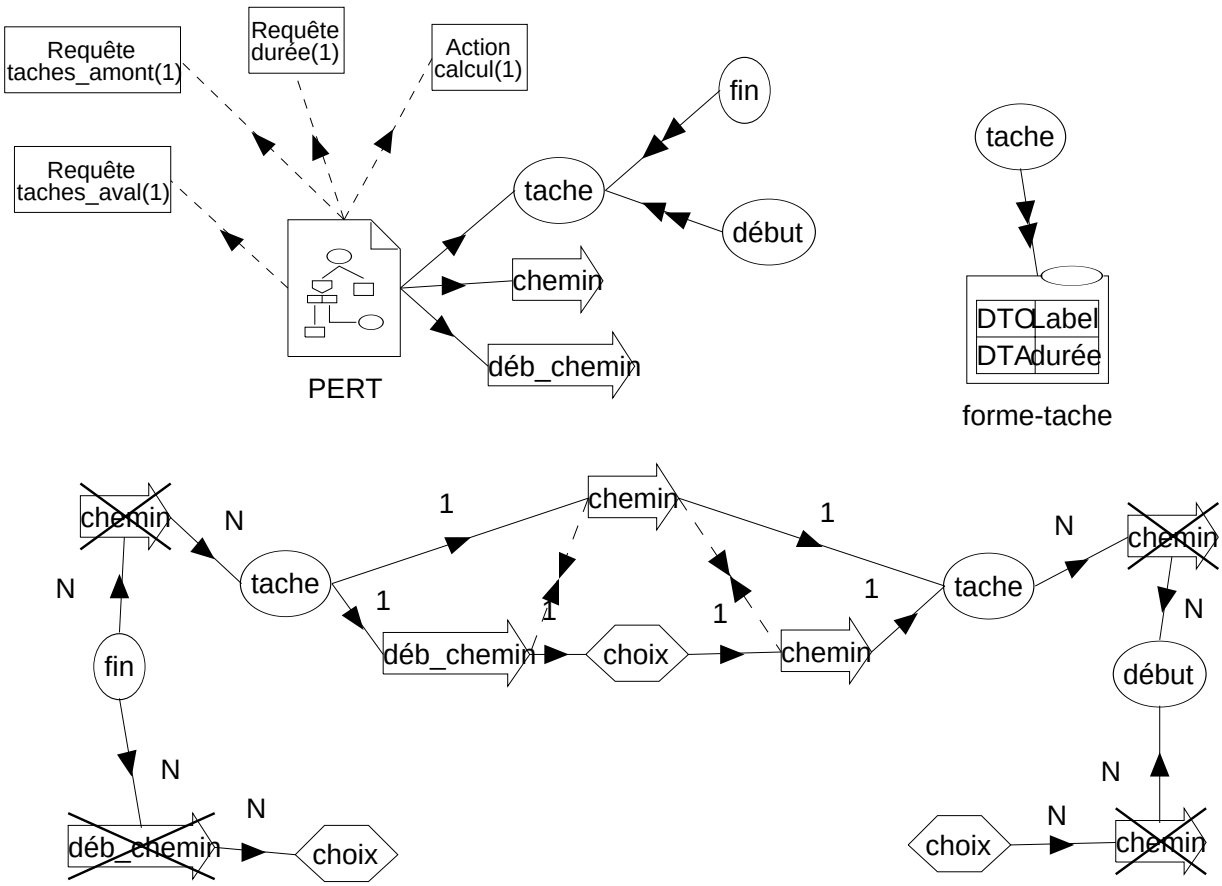


FIG. 1.4: *Extrait de la méta-modélisation GRAPHTALK d'un réseau PERT*

Le logiciel GRAPHTALK est destiné à la production et à l'utilisation **d'ateliers** de Génie logiciel. Ces **ateliers** sont produits à partir de la compilation de **méta-modèles** et permettent de dessiner et de gérer les modèles des systèmes. Ils permettent également de concevoir l'interface de construction des modèles et d'utilisation de la méthode, ou **hypergraphe**, mettant en œuvre successivement les différents modèles, les **graphes**.

La partie description des modèles permet de concevoir très rapidement un atelier permettant de dessiner des modèles dans un formalisme nouveau. Il est possible de spécifier les **classes d'objets et les liens autorisés**, ainsi que les **liens interdits** (voir figure 1.4). Ceci est très pratique, notamment lorsque des liens sont interdits pour une sous-classe alors qu'ils sont autorisés pour une classe d'objets. L'association de **formes** aux objets,

aux liens ou aux graphes permet de spécifier l'aspect graphique des modèles. La facilité d'utilisation et la rapidité de mise en œuvre de GRAPHTALK permettent de construire et de tester rapidement de nouveaux éditeurs.

Cependant, les modèles spécifiables sont des modèles statiques, même s'il est possible de spécifier des **requêtes** ou des **actions** associées à des instances d'objets ou à des menus. De plus, les méta-modèles manquent de **sémantique** dans la mesure où :

- les classes d'objets ne peuvent être liées que par des liens graphiques mais non par des liens purement sémantiques,
- les méta-liens ne sont pas nommés.

Des liens supplémentaires ont d'ailleurs été proposés dans [Niclet et al.96] pour intégrer l'étude **sémantique**, ainsi que l'utilisation conjointe de GRAPHTALK et d'un noyau de simulation développé en Smalltalk pour l'étude de la **dynamique**.

L'étude de la **syntaxe** d'un langage sans l'étude de la **sémantique** associée n'est pas suffisante pour assurer sa cohérence interne. L'utilisation de GRAPHTALK ne permet donc pas d'atteindre notre objectif d'étude rigoureuse et complète des activités de modélisation. Afin d'associer l'étude de la **sémantique** à l'étude de la **syntaxe**, d'autres travaux utilisent une approche données.

1.4.2 L'approche données

Pour améliorer la **syntaxe et la sémantique (lecture statique) du modèle Grafcet**, le groupe « Génie des Systèmes Intégrés de Production » du CRAN a choisi de construire des **méta-modèles** en NIAM [Habrias88] (figure 1.5). Un premier méta-modèle [Lhoste94] a permi de **formaliser** le **modèle** Grafcet défini dans [Afnor82]. Une extension de ce méta-modèle [Bon bierel94] a permis de prendre en compte de nouveaux **concepts** (macro-étape, grafcet global, partiel . . .) ou des **extensions** de concepts existants (étape, transition). Ces travaux ont ainsi prouvé que l'approche de méta-modélisation permet de pérenniser le Grafcet en assurant la cohérence de ses extensions. Ils montrent aussi qu'un modèle graphique n'est pas forcément lisible puisque le méta-modèle, de par sa taille, ne peut être lu que de façon fragmentaire.

Le **modèle** BASE-PTA [Afnor96] a pour objectif d'intégrer l'ensemble des **données** issues de toutes les phases du cycle de vie des SAP. Afin d'intégrer les données issues des phases de spécification à BASE-PTA, Florent Couffin [Couffin97] a construit un modèle de référence étendu pour le Génie Automatique. Il a été obtenu par intégration d'un **méta-modèle de référence** (figure 1.6) au modèle BASE-PTA. Le **formalisme** utilisé est

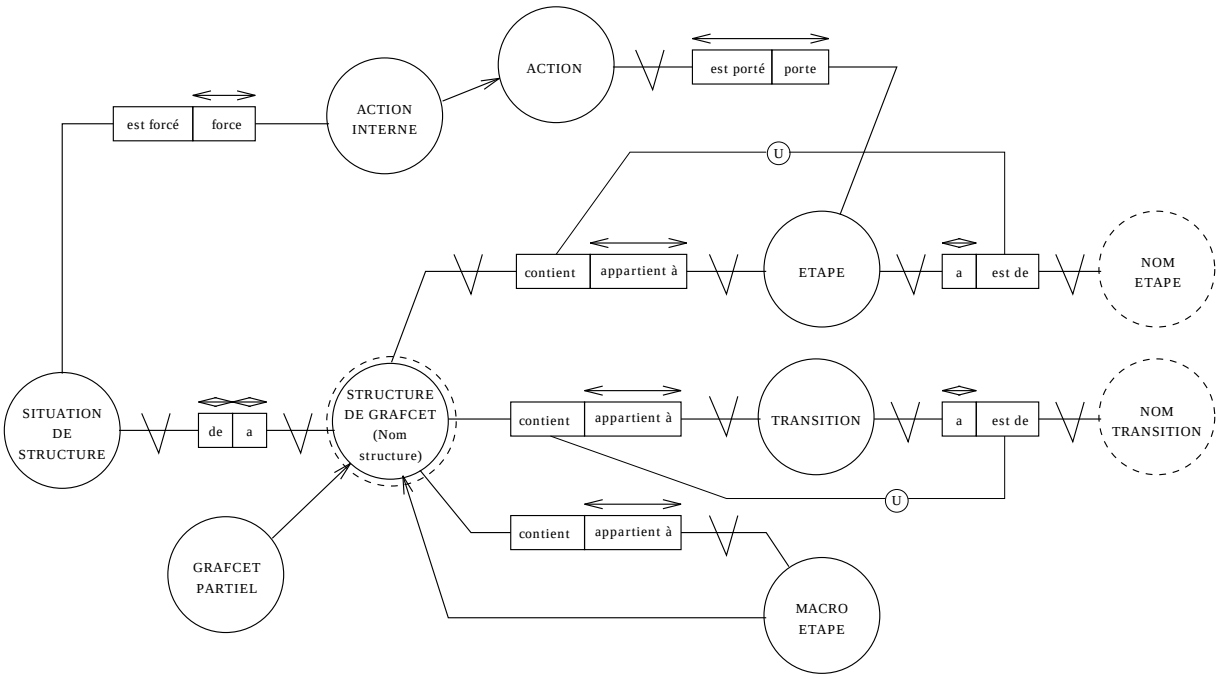


FIG. 1.5: *Extrait de la méta-modélisation en NIAM du Grafcet [Bon bierel94]*

OMD-GA, inspiré du **modèle** entité/relation, avec de nombreuses **extensions** : spécialisation/généralisation, relation d'agrégation, contraintes sur rôle et relations, contraintes explicites, relation d'identification généralisée. Le méta-modèle générique produit permet donc, par **spécialisation**, de construire le méta-modèle de tout outil de modélisation et d'étudier, par le modèle BASE-PTA, son intégration au sein d'un outil logiciel.

Ces approches données permettent une expression précise et claire de la syntaxe et de la sémantique des modèles. Cependant, beaucoup de modèles construits lors de la conception des SAP sont des modèles dynamiques. L'approche données fait l'impasse sur cet aspect dynamique, sur lequel au contraire l'approche traitement est centrée.

1.4.3 L'approche traitements

Une modélisation algébrique a été utilisée pour améliorer la **sémantique « interne » (lecture dynamique) du modèle Grafcet** [Bon bierel94]. Le principe de cette approche est d'utiliser des équations algébriques pour caractériser l'état de chaque étape en fonction des conditions d'activation et de désactivation de cette étape. Cette modélisation permet de bien formaliser le **comportement dynamique** du Grafcet. Cependant, la spécification du comportement dynamique ne peut être comprise qu'à condition que la spécification de l'aspect statique soit acquise. L'absence de description des données, dans ce méta-modèle, nous fait penser qu'il ne peut venir qu'en complément, d'un autre méta-modèle. Malgré

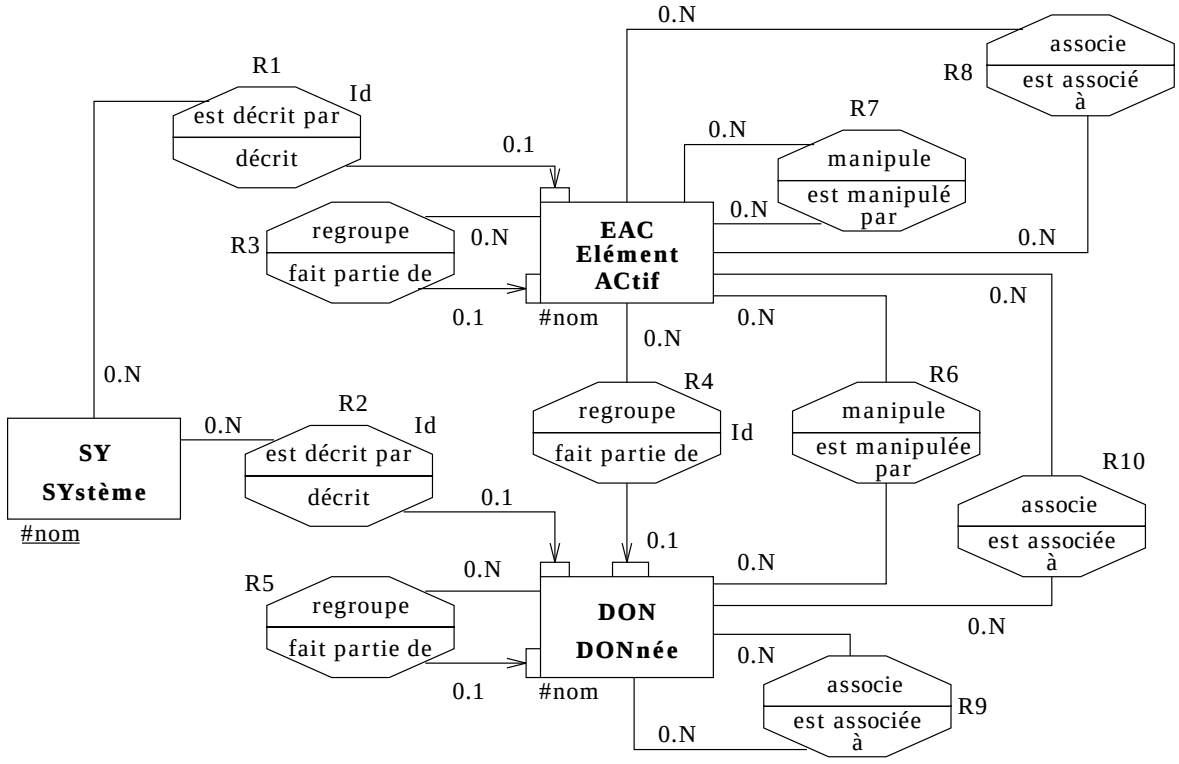


FIG. 1.6: *Méta-modèle de référence des outils de modélisation pour le Génie Automatique [Couffin97]*

l'intérêt démontré de l'utilisation de deux méta-modèles distincts, pour la spécification des aspects statiques et dynamiques, nous pensons que ceci peut conduire à la construction de méta-modèles ne prenant pas en compte toute la sémantique des concepts modélisés et donc, finalement, entraîner des pertes ou des erreurs sémantiques importantes.

Toutes les approches de méta-modélisation présentées jusqu'à présent ne s'intéressent qu'à des aspects des modèles construits, sans préciser comment les construire. Il semble pourtant difficile d'assurer la rigueur et la cohérence du résultat sans se soucier du moyen utilisé. Les travaux présentés maintenant s'intéressent à ces étapes de construction.

1.4.4 La construction des modèles

EXPRESS est un **langage** normalisé [Iso93] de spécification pour l'échange de données. [Pierra et al.95] montre qu'il peut être utilisé pour la conception d'un système de XAO, aussi bien pour la **base de données, le modèle** du système, que pour les **programmes** qui l'exploite. L'exemple traité est la spécification d'un **graphe** et sa spécialisation en **réseau** PERT (figure 1.7). La notion de schéma d'EXPRESS permet de spécifier le contenu et les propriétés d'un réseau PERT sans se préoccuper des programmes de création, mo-

```
SCHEMA PERT_final_schema ;
REFERENCE FROM graph_schema (graph, sommet, arc) ;
ENTITY reseau_PERT
SUBTYPE OF graph ;
   entree : etape_debut ;
   sortie : etape_debut ;
   noeuds : SET [0:?] OF etape\_interieur ;
DERIVE
SELF\graph.sommet : SET [1:?] OF etape:=[entree] + [sortie] + noeuds ;
SELF\graph.arc : SET [0:?] OF tache ;
WHERE
   acyclic : is_acyclic (SELF\graph.sommet, SELF\graph.arc) ;
END_ENTITY ;
...}
```

FIG. 1.7: *Spécialisation d'un graphe en un réseau PERT [Pierra et al.95]*

dification ... ce qui assure une séparation entre les données et les traitements. Il n'y a pourtant pas indépendance entre les deux puisque ces traitements sont aussi spécifiés en EXPRESS. De plus, EXPRESS présente l'avantage de permettre un prototypage rapide des spécifications (classe C++) ou un codage aisé. L'utilisation d'EXPRESS pour la spécification permet ainsi d'assurer que les systèmes de XAO produits respecteront bien la spécification du modèle.

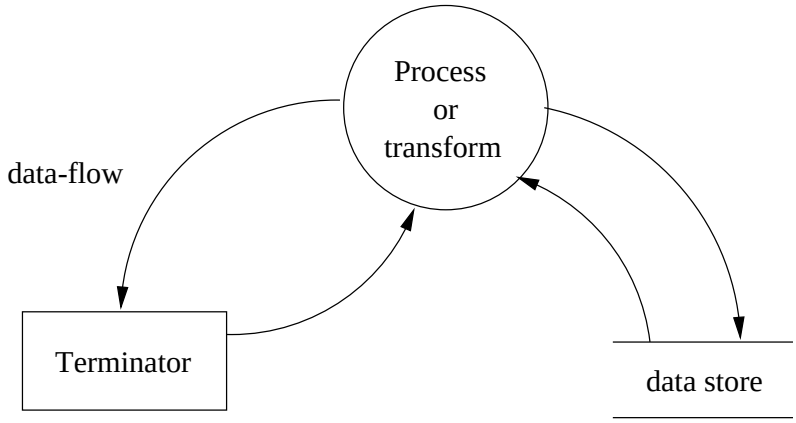


FIG. 1.8: *Exemple de diagramme de flots de données (DFD) [Gee95]*

David M. Gee étudie la **spécification formelle de la syntaxe des langages visuels**. Dans [Gee95] il utilise le **langage** Z pour la spécification des DFD de SA-RT de Yourdon et les DSD de Jackson, vus tous les deux comme des cas particuliers de **graphes**

(sur la figure 1.8), « terminator », « Process or Transform » et « data store » sont des nœuds, les « data-flows » sont des arcs). Cet auteur étudie non seulement les **concepts** présents dans ces graphes, mais également leur **forme**. A travers les **contraintes** relationnelles de ces graphes il montre que certaines contraintes doivent toujours être vérifiées (par exemple: un nœud ne peut être relié qu'à un arc), alors que d'autres sont transgressées durant la phase de construction des modèles (par exemple: un nœud doit être relié à au moins un arc, ce qui n'est pas vrai au moment où le nœud est créé). La modélisation des structures hiérarchiques telles que les Statecharts est également étudiée. Au contraire de David M. Gee, A. J. Galloway et S. J. O'Brien [Galloway et al.94] n'étudient que la **syntaxe abstraite** (indépendante des caractéristiques typographiques, telles que les caractères et conventions d'écriture qui forment **la syntaxe concrète**) des modèles essentiels de SA-RT de Ward et Mellor. Le **langage** utilisé est également Z (figure 1.9). Les auteurs considèrent eux que la construction de modèles est libre de contraintes, puisqu'ils font la distinction entre des modèles et des modèles **validés**, c'est-à-dire qui respectent un certains nombre de contraintes. Dans ces travaux, l'utilisation de Z permet à la fois d'étudier la structure des langages étudiés à l'aide de notions mathématiques (théorie des ensembles, logique des prédicats) mais aussi d'exprimer les règles de construction de modèles, le tout dans une structure cohérente.

$$
\begin{array}{|l}
\hline
DFD\_TYPE \\
title : IDENT \\
terminators : \mathbb{F}\ TERM\_TYPE \\
dataflows : \mathbb{F}\ DF\_TYPE \\
eventflows : \mathbb{F}\ EF\_TYPE \\
system : NODE \\
\hline
\end{array}
$$

FIG. 1.9: *Schéma Z d'un DFD [Galloway et al.94]*

Ces travaux permettent de spécifier de quelle manière sont construits les modèles tout en assurant le respect de leur syntaxe. De plus, l'utilisation d'un langage formel, tel que Z, permet de valider et de vérifier le méta-modèle construit. Cependant, dans la conception d'un SAP, un modèle n'est pas isolé des modèles précédents. Cette intégration des modèles doit aussi être étudiée pour la cohérence des différentes activités de conception et donc des SAP étudiés.

1.4.5 L'intégration des modèles

Les travaux du LURPA

De nombreux travaux ont également eu lieu au LURPA sur la méta-modélisation [Lesage94] au sein de l'équipe Conception des Systèmes Avancés de Production. Pour réaliser les **méta-modèles**, le **formalisme** retenu a été le formalisme entité/relation jugé bien adapté à la représentation de la **syntaxe** et de la **sémantique** des **techniques de modélisation**. Ces travaux précurseurs ont amené les auteurs de [Denis et al.93] à définir la méta-modélisation :

> « Modéliser une **méthode de conception**, c'est donc :
>
> « – modéliser chacune des **techniques de modélisation retenues (essentiellement la syntaxe des modèles)**,
>
> « – modéliser les **techniques de construction de modèles (sémantiques associées aux modèles)**,
>
> « – modéliser les **techniques de passage d'un modèle à un autre (l'aspect "intégration" de la méthode)**.
>
> Nous appelons méta-modèle le modèle global, ainsi constitué, d'une technique de modélisation ou d'une méthode de conception. »

Cette définition permet de façon évidente de bien montrer l'originalité des travaux réalisés au sein du LURPA. Les travaux qui y sont réalisés ne portent en effet pas uniquement sur la définition des techniques de modélisation, mais également sur leur intégration au sein des méthodes.

Dans ses travaux de thèse, Bruno Denis [Denis94] montre ainsi tout l'intérêt de la méta-modélisation pour définir la syntaxe et la sémantique de modèles construits dans un formalisme nouveau (modèle d'implantation) ou dans un formalisme déjà connu (SA-RT). A partir de ces deux méta-modèles, il en construit un troisième qui montre l'intégration entre ces deux formalismes. L'intégration consiste alors à définir des relations entre des entités issues des deux méta-modèles existants, aucune nouvelle entité n'étant créée.

Par ailleurs, [Lesage et al.92] démontre par l'exemple (la méta-modélisation de la méthode intégrée de passage d'un MCT à un MOT) comment les méta-modèles peuvent être utilisés pour la définition des méthodes. Les méthodes sont envisagées suivant deux aspects : les techniques de construction des modèles d'une part, et l'intégration des différents modèles au sein d'un projet d'autre part. Ainsi, les entités « règle de gestion » et « action » (figure 1.10) sont des entités ne faisant pas partie des modèles produits mais de la technique de construction des modèles. Chacune de ces entités participe à l'intégration

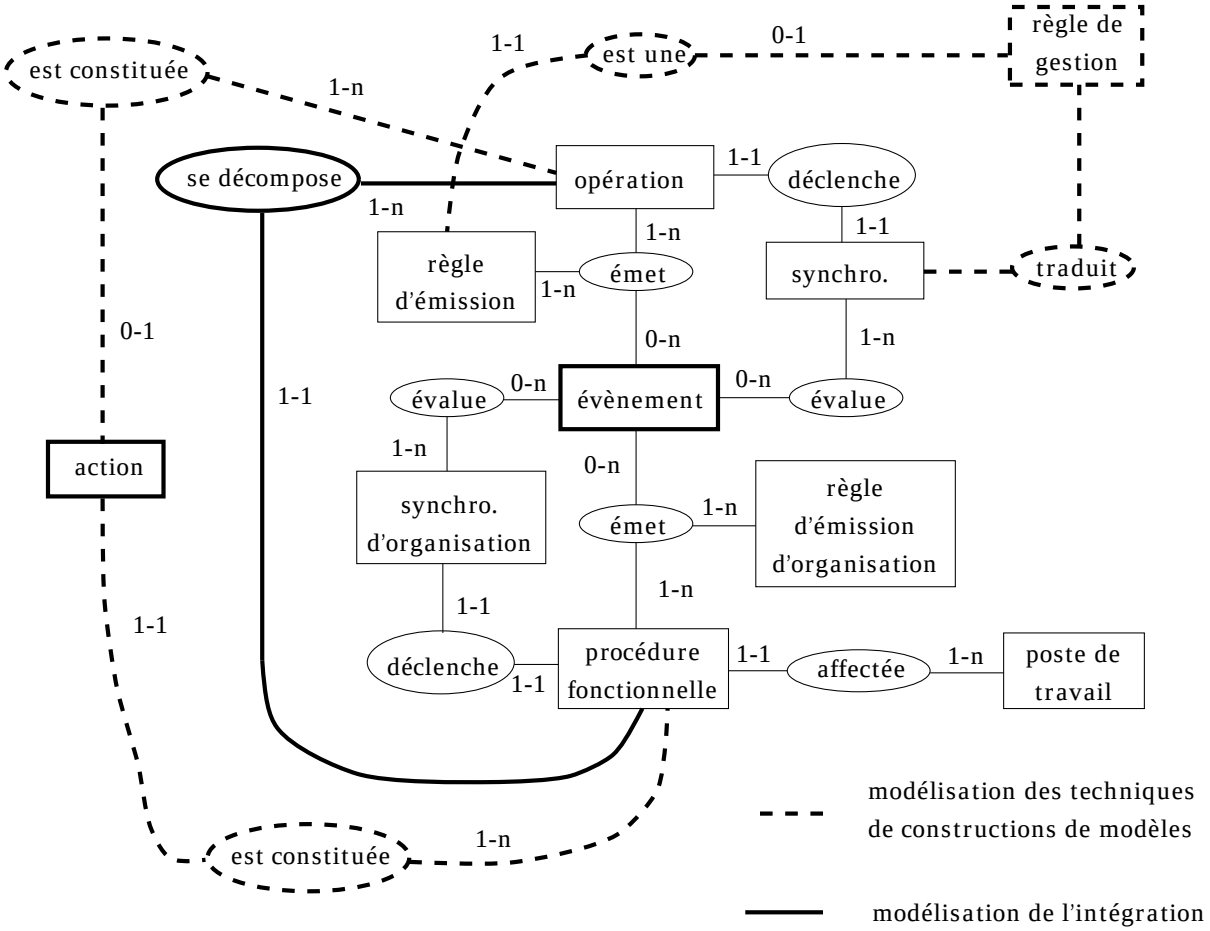


FIG. 1.10: *Méta-modèle partiel de la méthode intégrée de passage d'un MCT à un MOT [Lesage et al.92]*

des deux modèles, soit parce qu'elle fait partie des deux, soit parce qu'elle participe à la mise en relation d'entités des deux modèles.

La limite majeure de cette approche est sa vision purement statique des méthodes. Le choix du formalisme entité/relation occulte les phases de construction ou de traduction de modèles qui sont des aspects primordiaux des méthodes.

La thèse de François Kiefer

Les travaux de François Kiefer [Kiefer96] ont eu pour cadre l'atelier de prototypes de la société AÉROSPATIALE, à Chatillon (92). Confronté à la multiplicité et à la diversité des méthodes de conception des **Systèmes Intégrés de Production** (SIP), F. Kiefer a étudié les différents cas possibles **d'intégration de méthodes**. La diversité des méthodes utilisées l'ont amené à réfléchir à une intégration par l'intermédiaire d'un **langage pivot**. Le choix d'un langage pivot a permis de limiter le nombre d'intégrations à étudier. Le

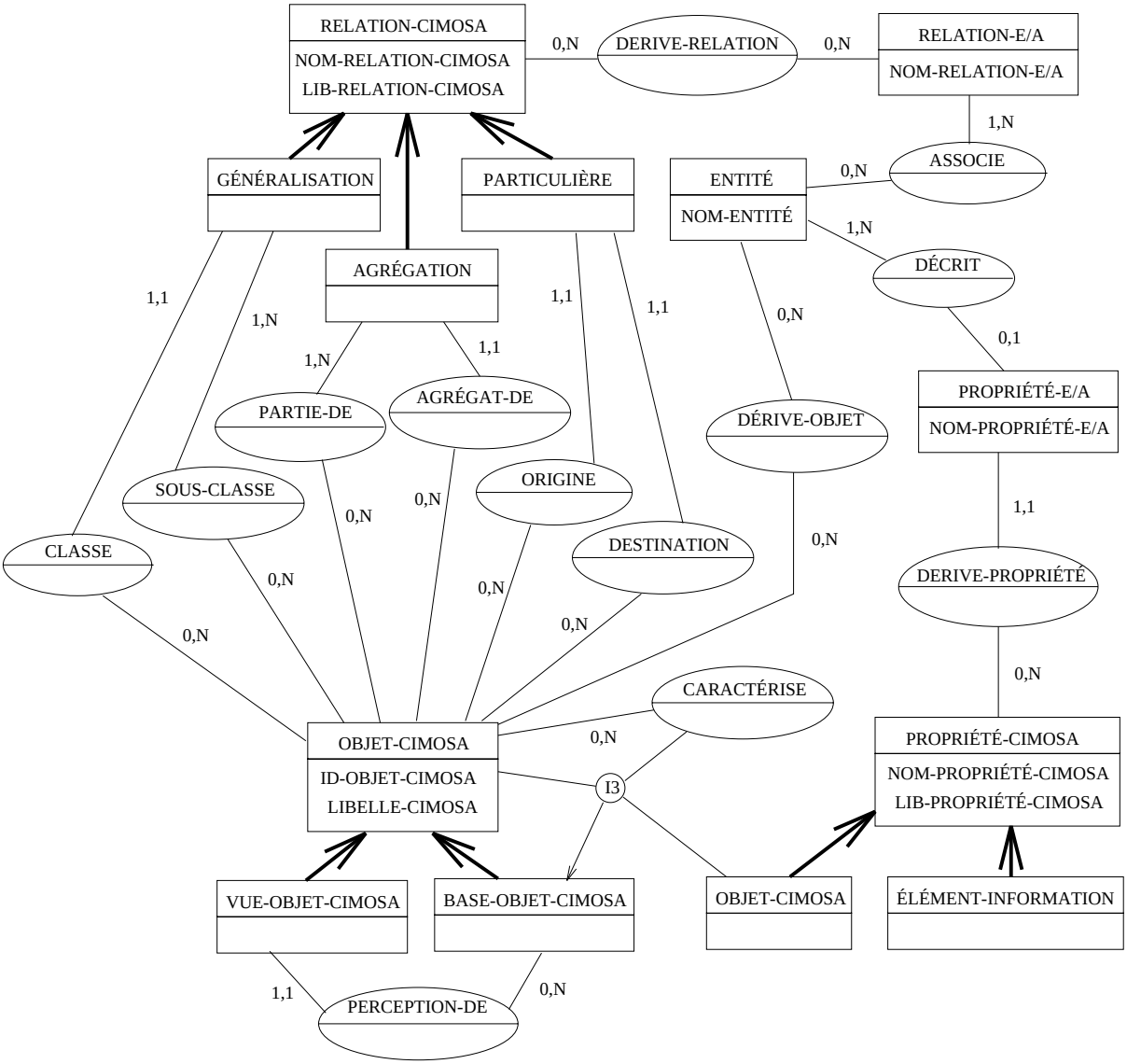


FIG. 1.11: *Méta-modèle de la vue information de CIM-OSA [Kiefer96]*

choix de CIM-OSA comme pivot a permis d'avoir une couverture complète des différentes **phases** d'étude des SIP. La nombre important de méthodes étudiées lui a permis de mettre en évidence dix cas de correspondance entre une méthode et le langage pivot. Ces dix cas correspondent à la fois à un travail sur la **sémantique** mais également au type **d'action** réalisable. Entité/relation est utilisé pour la construction des méta-modèles. Par contre, les **règles** d'importation et d'exportation sont exprimées en français, ce qui malheureusement ne permet pas de vérifier leur cohérence interne et encore moins leur cohérence par rapport aux méta-modèles.

Le travail réalisé est à notre avis le plus complet et le plus directement applicable dans le domaine de l'intégration de méthodes. Le choix de entité/relation pour construire des

méta-modèles à but pédagogique et pragmatique, comme support efficace de communication au sein de l'entreprise, était très pertinent. Par contre, il s'est avéré un handicap pour la phase d'expression des règles de passage d'un modèle à un autre en rendant cette description textuelle indépendante de la description des différents modèles. Comme nous l'avons déjà dit, ce choix empêche aussi la modélisation du comportement dynamique des modèles.

1.5 Bilan des travaux existants

A travers les différents travaux que nous avons présentés dans ce chapitre nous avons conduit deux études.

La première étude est celle du vocabulaire utilisé pour la modélisation des SAP. Les termes outil, langage, approche, modèle, diagramme fonctionnel, formalisme ... désignent le système utilisé pour modéliser. Classe d'objet, lien, requête, action, règle, concept, forme ... sont les éléments de ce système. Cette variété est due à un manque de définitions précises des différents aspects de la modélisation. Elle entraîne un défaut de rigueur dans l'activité de modélisation.

La seconde étude concerne la cohérence des activités de modélisation et plus particulièrement l'étude de ces activités par méta-modélisation. Les travaux de méta-modélisation utilisent des approches dédiées à quelques aspects des modèles : syntaxe pour GRAPH-TALK, statique pour [Lhoste94] [Couffin97], dynamique pour [Bon bierel94], syntaxe et construction de modèles pour [Pierra et al.95] [Gee95], intégration de modèles pour [Denis et al.93] [Kiefer96]. Chacun de ces travaux permet, grâce à la méta-modélisation, de clarifier un aspect des langages de conception. Cependant, en n'étudiant pas tous les aspects de la modélisation, ils n'imposent pas d'en définir clairement tous les concepts.

Notre objectif est, au contraire, de proposer une approche permettant de définir avec rigueur tous les aspects d'un langage ou d'une méthode. Le chapitre deux sera donc tout d'abord consacré à la définition rigoureuse des différents aspects de la modélisation. Une fois ces bases posées, nous pourrons présenter l'ensemble des besoins liés à la définition des activités de modélisation. Nous ferons ensuite de même pour les besoins liés à l'intégration des activités de modélisation.

Chapitre 2

Langages, méthodes et méta-modèles

Notre objectif est d'améliorer la qualité des SAP en améliorant la rigueur et la cohérence des différentes activités de modélisation de ces SAP. Le chapitre précédent nous a permis de mettre en évidence la diversité du vocabulaire utilisé dans ces activités de modélisation. Cette diversité est le reflet d'un manque de rigueur des termes utilisés et de leur définition. Ce chapitre commencera donc par l'étude de l'activité de conception pour définir les trois concepts qui en forment l'ossature : le modèle, le langage et la méthode. Ces définitions devront être rigoureuses et cohérentes entre elles, c'est pourquoi elles seront données par « affinages » successifs. Les définitions données ne devront donc pas être tout d'abord considérées comme définitives mais simplement cohérentes, vis-à-vis des définitions précédentes, à l'instant considéré du discours. Lorsque ces définitions seront définitives, elles apparaîtront en caractères gras dans le texte.

Dans le chapitre précédent, nous avons également montré que différents aspects des langages et des méthodes peuvent être définis à travers les méta-modèles. Cependant, aucun des travaux présentés ne spécifiait l'ensemble des aspects des langages et des méthodes. Notre objectif est, au contraire, d'étudier en même temps l'ensemble de ces aspects afin d'être le plus complet et cohérent possible. La section 2 sera donc consacrée aux différents aspects de l'étude de la syntaxe et de la sémantique d'un langage. La section 3 sera consacrée à l'étude de toutes les méthodes utilisant ces langages : construction, intégration, validation et jeu de modèles. Enfin, dans la section 4, nous apporterons quelques précisions sur le concept de méta-modélisation.

2.1 L'activité de conception

L'activité de conception d'un SAP peut être représentée [Couffin97] comme une activité de résolution d'un problème c'est-à-dire de passage d'un problème modélisé vers une solution modélisée (figure 2.1). Cette activité de conception est décomposable [Wilson86]

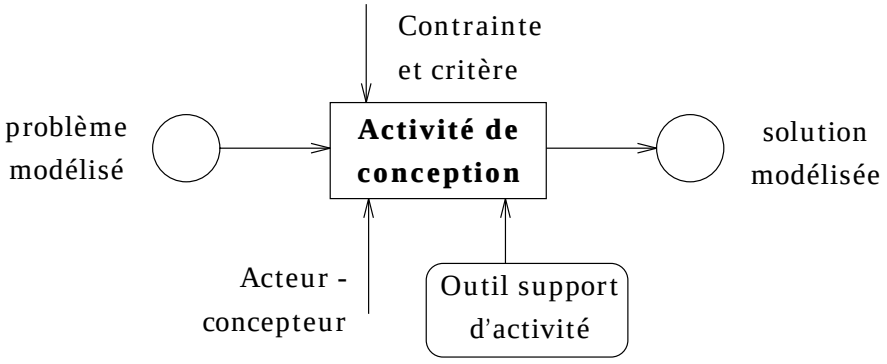


FIG. 2.1: *Représentation simplifiée de l'activité de conception [Couffin97]*

en deux sous-ensembles couplés :

- un système d'activités, représentatif des méthodes, techniques, ou d'une méthodologie,
- un système social, représentatif des participants à l'activité de conception.

Nous n'étudierons ici que le système d'activités, pour lequel nous allons analyser ses différents constituants : les méthodes, langages et modèles construits. Lorsque nous aurons à étudier un constituant, nous présenterons les différentes définitions existantes, et nous les commenterons. Pour établir nos propres définitions, nous nous sommes fixés les critères suivants :

- si il y a unanimité sur une définition, nous adoptons cette définition,
- sinon, par ordre de priorité :
  1. la définition doit être cohérente par rapport à la définition des autres concepts,
  2. la définition doit être cohérente par rapport à des définitions existantes dans d'autres domaines scientifiques,
  3. la définition la plus souvent utilisée en Génie Automatique l'emporte.

Le modèle représentant à la fois le résultat mais aussi le point de départ de cette activité, notre étude commence par la notion de modèle.

2.1.1 Qu'est-ce qu'un modèle ?

Différents auteurs proposent les définitions suivantes :

Modèle [Calvez90] [Morel92] : un modèle est une interprétation explicite par son utilisateur de la compréhension d'une situation, ou plus simplement d'une idée qu'il

se fait sur la situation. Il peut être exprimé par des mathématiques, des symboles, des mots, mais essentiellement, c'est une description d'entités et de relations entre elles. Ainsi, modéliser revient à élaborer une vue partielle plus ou moins abstraite de l'existant ;

Modèle [Lhoste94] : vue partielle d'un système appréhendé à travers une théorie cadre pour le(s) point(s) de vue de modélisation, éventuellement associé à une méthode, et exprimé par un langage (moyen d'exprimer un modèle pour permettre son exploitation ou la communication du message qu'il porte) ;

Modèle [Couffin97] : représentation de l'objet de la conception. Les primitives et les règles qui permettent d'interpréter et de construire ces modèles sont appelées Modèle ;

Dans le cadre de la modélisation des SAP, les modèles doivent avoir une existence concrète, les modèles abstraits (c'est à dire les « vues de l'esprit ») ne pouvant pas être échangés entre concepteurs ne sont pas pris en compte. Dans les trois définitions présentées plus haut, et qui sont assez proches les unes des autres, un modèle est présenté soit comme une vue soit comme une représentation. Parmi ces deux termes, « représentation » semble mieux exprimer ce côté concret du modèle, le terme « vue » pouvant plus facilement se référer à un modèle abstrait. De plus, le terme système est plus général que celui d'objet, qui a un sens particulier en informatique. Nous inspirant de ces définitions, nous proposons donc une première définition de modèle :

Modèle :	*représentation d'un système ou d'une famille de systèmes. Il peut être exprimé par des mathématiques, des symboles, des mots.*

Un modèle peut représenter un système ou une famille de système car il existe des modèles plus ou moins génériques d'un système. Le degré de généricité correspondant à la taille de la famille des systèmes modélisés. Cette famille correspond à l'ensemble des systèmes qui ont des propriétés, des fonctions, des comportements ... communs qui sont justement l'objet du modèle (figure 2.2).

2.1.2 Avec quoi construire un modèle ?

Les trois définitions précédentes de modèle définissent également le moyen utilisé pour écrire un modèle. [Calvez90] et [Morel92] décrivent un modèle comme un ensemble d'entités et de relations. Cette définition correspond à la description de modèles par méta-modèles réalisés avec le formalisme entité/relation. Nous ne la retiendrons pas car nous lui préférerons une définition indépendante des moyens de méta-modélisation utilisés. Dans

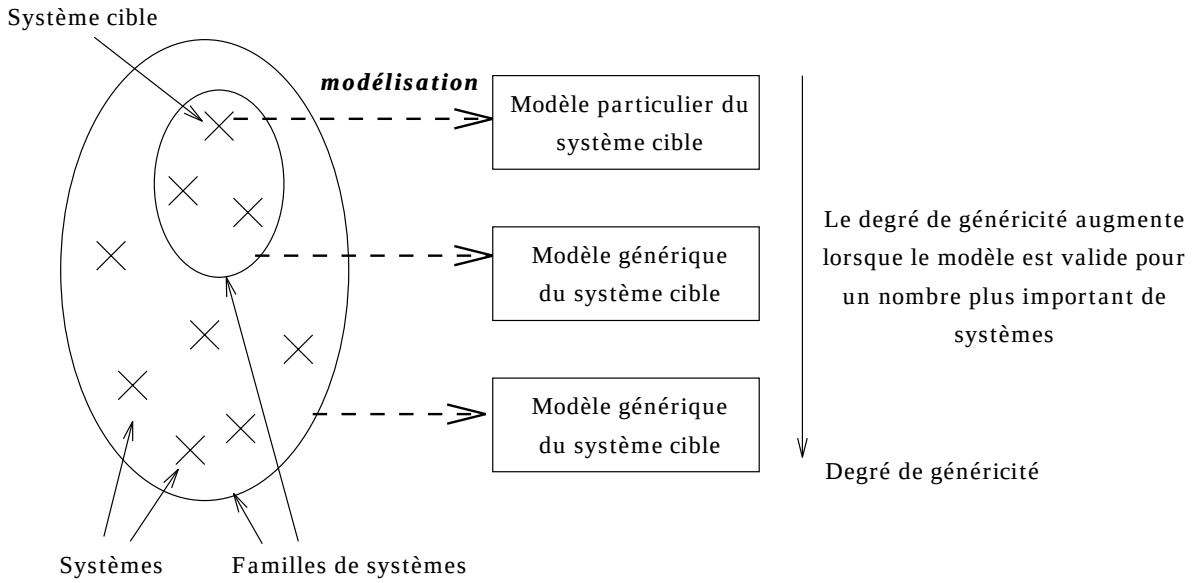


FIG. 2.2: *Degré de généricité d'un modèle*

[Couffin97], les modèles sont construits à partir d'un Modèle . Nous ne retiendrons pas non plus cette définition car nous préférerons éviter d'utiliser des distinctions purement typologiques entre des notions différentes. [Lhoste94] exprime un modèle par l'intermédiaire d'un langage, de même que [Lher et al.96]. Cette approche nous paraît cohérente car elle ne pose pas de problème d'interprétation et est conforme à celles utilisées en informatique et en linguistique. Nous retiendrons donc le terme langage que nous définirons pour le moment de façon assez générale par :

Langage : système de représentation.

De plus afin de montrer le lien entre langage et modèle, nous proposons de faire évoluer la définition précédente de modèle :

Modèle : résultat de l'utilisation d'un langage pour la modélisation d'un système ou d'une famille de systèmes. On parlera de l'instanciation d'un langage pour la modélisation d'un système ou d'une famille de systèmes.

Le terme outil est aussi utilisé, soit pour désigner l'outil de modélisation [Couffin et al.96], ou pour désigner le logiciel utilisé [Bodart et al.96]. La première signification du terme outil correspond à notre choix du terme langage. Quant à la deuxième, elle ne nous parait pas utile car nous considérons que l'outil utilisé pour mettre en œuvre un langage est toujours un logiciel.

2.1.3 Comment définir les éléments d'un langage ?

Le langage est utilisé pour construire des modèles. Le langage a donc les propriétés de l'ensemble des modèles construits avec ce langage. Définir un langage consiste donc à définir l'ensemble des modèles constructibles avec ce langage.

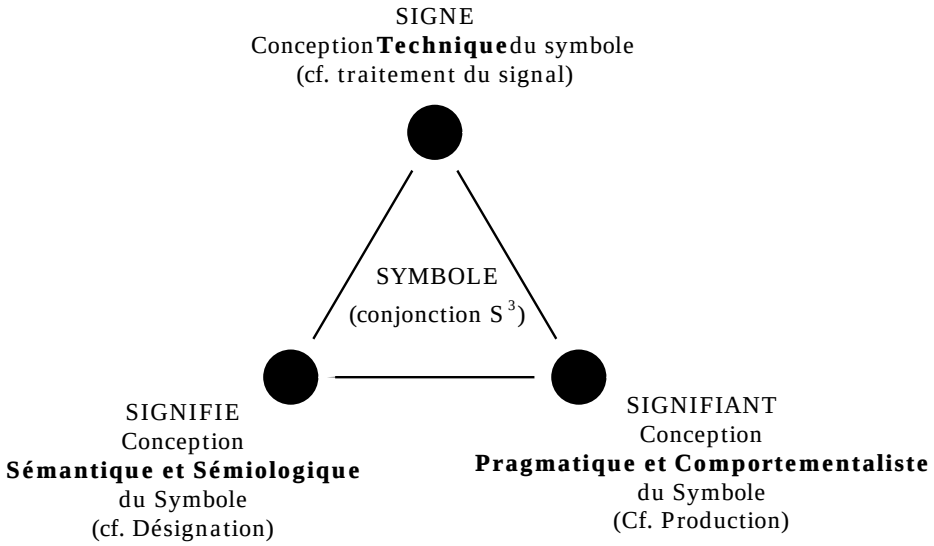


FIG. 2.3: *Modèle canonique de l'information [Lemoigne90]*

Le modèle canonique de l'information (conception systémique de l'information présenté dans [Lemoigne90]) présente une information comme une configuration stable de symboles : « Le symbole est un opérateur - récursif à point fixe, assurant les fonctions de symboles et de production de symboles. Il s'exprime par la conjonction S3 ». Le symbole y est vu comme la composition du signe, du signifiant et du signifié (figure 2.3). Le signe est la représentation du symbole : lettres, figures géométriques, dessins. Le signifié est la désignation du symbole. Le signifiant représente son sens.

François Kiefer [Kiefer96] envisage le modèle comme un ensemble de concepts et de formalismes (figure 2.4). Les formalismes sont le moyen de représentation des concepts. Il existe donc des relations entre les concepts et les formalismes. En outre, les formalismes ont des relations entre eux qui sont la matérialisation des relations inter concepts. Le modèle de François Kiefer est très proche de la conception analytique de l'information qui identifie celle-ci par un doublet signe/signification qui peuvent être traités indépendamment.

Que ce soit dans le modèle analytique ou dans le modèle de Kiefer, la signification englobe le signifié et le signifiant. En effet, il ne semble pas possible d'étudier les signifiants d'un modèle sans désigner, et donc utiliser les signifiés, les symboles utilisés pour construire ce modèle. Par contre, il peut être intéressant d'étudier les signifiés d'un langage, sans pour autant se soucier des signes ou des signifiants (nous verrons plus tard que cela correspond

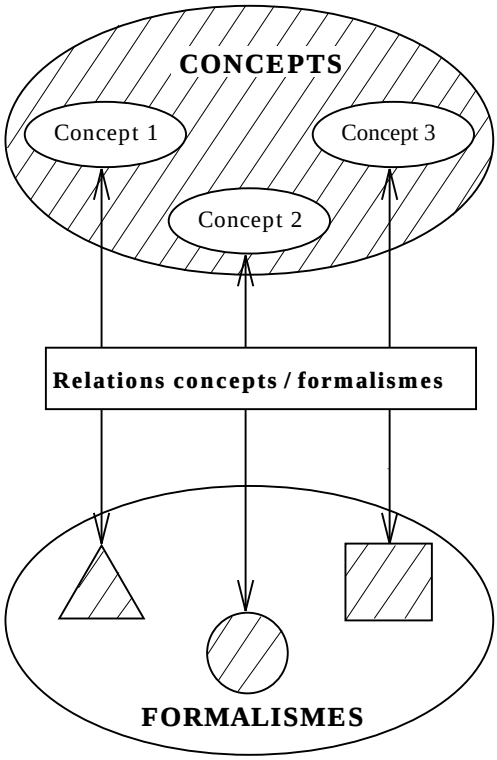


FIG. 2.4: *Composition d'un modèle d'après [Kiefer96]*

à une partie de l'analyse de la syntaxe d'un langage). Le modèle canonique semble donc être le plus détaillé quant à l'étude des symboles et nous permet de proposer la définition suivante (afin de rendre notre discours plus lisible, nous utiliserons parfois les termes entre parenthèses) :

Symbole :	***conjonction d'un signe (représentation) capable d'être à la fois signifié (désignation) et signifiant (sens) [Lemoigne90].***

Cette définition nous permet à nouveau de proposer de nouvelles définitions de langage et de modèle, afin d'assurer la cohérence de toutes nos définitions :

Langage :	*système de représentation symbolique. Définir un langage consiste à définir l'ensemble des modèles constructibles avec ce langage.*

Modèle :	*instanciation d'un langage pour la représentation d'un système.*

Construire un modèle consiste donc à instancier des symboles. La combinaison de plusieurs symboles est encore un symbole. Une des difficultés de la définition d'un langage sera donc de distinguer ce qui dans le langage est un symbole de ce qui est une combinaison de symboles.

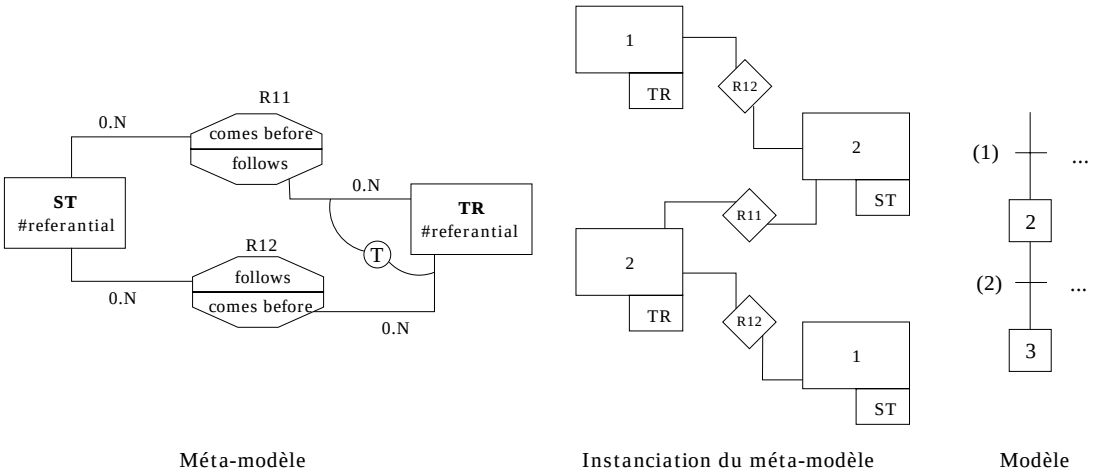


FIG. 2.5: *Exemple d'instanciation de symboles et de liens [Couffin et al.98]*

Dans la pratique, cette distinction entre niveaux de symboles est du domaine de la connaissance métier des langages utilisés pour la conception des SAP. Par exemple, tous les travaux de méta-modélisation du grafcet [Lhoste et al.93] [Bon bierel98] [Couffin et al.98] considèrent que les étapes et les transitions sont des symboles. [Couffin et al.98] méta-modélise les transitions et les étapes par des entités, et les arcs par des relations. Dans tous les méta-modèles du grafcet, les arcs reliant les étapes aux transitions ne sont pas considérés comme des symboles, car ils ne sont définis qu'à travers les étapes et les transitions qu'ils relient. Ces arcs sont des liens entre les symboles du langage grafcet. Ils permettent de structurer le langage. Nous définirons donc un langage par ses symboles et par les liens existants entre symboles. Nous avons choisi le terme lien pour éviter l'amalgame avec les relations du langage entité/relation. Nous pouvons donc compléter la définition de langage par :

Langage :	***système de représentation symbolique. Définir un langage consiste à définir l'ensemble des modèles constructibles avec ce langage. Ceci consiste à décrire les symboles et les liens entre symboles de ce langage.***

2.1.4 Comment obtenir une solution modélisée ?

Certes, un langage est indispensable pour construire un modèle, mais connaître un langage ne signifie pas savoir s'en servir pour modéliser. Pour aider l'analyste, une méthode peut être associée à un langage. La méthode peut être alors définie comme :

Méthode [Lhoste94] : processus menant à la création d'un modèle.

Méthode [Kiefer96], d'après [Rolland et al.90] : le premier élément d'une méthode est l'outil de modélisation associé à cette méthode. Le second élément est la démarche de travail à mettre en œuvre autour de cet outil pour parvenir à ses fins.

Dans les premières phases de la conception d'un SAP, le problème à résoudre n'est souvent exposé qu'au travers d'un cahier des charges textuel, mal structuré, en bref informel. L'activité de conception peut donc être vue comme une activité d'utilisation d'une méthode de transformation d'une « page blanche » en un modèle. Par contre, au fur et à mesure de l'avancée de la conception, de plus en plus d'informations peuvent être extraites des modèles amont pour aider ou contraindre la construction de nouveaux modèles. La définition suivante permet de bien faire la distinction entre les langages utilisés, les techniques de construction de modèles (indépendamment des modèles amont) et les techniques de passage d'un modèle à un autre.

Méthode (d'après [Denis et al.93]) : modéliser une méthode de conception, c'est :

- modéliser chacune des techniques de modélisation retenues,
- modéliser les techniques de construction de modèles,
- modéliser les techniques de passage d'un modèle à un autre.

Ces deux derniers types de techniques permettent de construire des modèles qu'il peut être nécessaire de vérifier, de valider ou de simuler. La définition du terme méthode doit donc être plus générale que les définitions précédentes. Les définitions suivantes le sont :

Méthode [Calvez90] [Morel92] : une méthode ou technique de résolution est caractérisée par un ensemble de règles bien définies qui conduisent pour le problème à une solution correcte.

Méthode [Gallimard92] : ensemble de règles et de démarches adoptées pour conduire une recherche.

Ces définitions mettent l'accent sur les règles permettant d'aboutir à la solution, dans notre cas un modèle. Finalement, les points importants qui apparaissent dans toutes les définitions précédentes sont :

1. une méthode comporte des langages utilisés pour construire des modèles (le problème initial, les solutions intermédiaires et la solution finale) ;
2. une méthode comporte une démarche de travail permettant de passer du problème exprimé à la solution modélisée ;

3. cette démarche peut permettre de construire des modèles, de passer d'un modèle à un autre, de valider un modèle, de vérifier un modèle, de simuler un modèle ... ;

4. cette démarche est constituée de différentes opérations soumises à des règles.

Nous proposons donc comme définition :

Méthode :	***une méthode est caractérisée par un ensemble de langages, et par une démarche permettant de passer d'un problème à une solution modélisée correcte. Cette démarche est constituée d'opérations soumises à des règles.***

2.2 L'étude d'un langage

2.2.1 L'étude de la syntaxe

Syntaxe abstraite et syntaxe concrète

Le dictionnaire de la langue du XIXe et du XXe siècle [Gallimard92] propose plusieurs définitions de la syntaxe :

Syntaxe (grammaire) : partie de la grammaire traditionnelle qui étudie les relations entre les mots constituant une proposition ou une phrase, leurs combinaisons, et les règles qui président à ces relations, à ces combinaisons.

Syntaxe (sémiotique) : ensemble de relations, de combinaisons de signes entre eux.

La première définition, bien qu'elle ne soit applicable qu'à du texte, est très proche de l'utilisation courante du terme syntaxe dans le domaine de l'étude des langages de conception des SAP, c'est à dire l'étude de la structure des modèles. Quant à la deuxième définition, elle nous ramène à la distinction utilisée dans [Galloway et al.94] entre syntaxe abstraite et la syntaxe concrète. Cette distinction est importante car elle permet de faire le lien entre les définitions de langage et de symbole et les différents travaux sur la syntaxe des langages. Les définitions suivantes permettent de bien mettre en évidence les rapports entre structure des modèles construits, l'étude des signes et l'étude des signifiés :

Syntaxe abstraite :	***ensemble des structures de signifiés composant un langage.***
Syntaxe concrète :	***ensemble des structures de signes composant un langage.***

Par rapport à ces deux définitions, il est facile de scinder les différents travaux portant sur l'étude de la syntaxe en deux groupes :

- d'une part, les travaux ne portant que sur l'étude de la syntaxe abstraite, c'est le cas de la plupart des travaux,

- d'autre part, les travaux portant également sur la syntaxe concrète : c'est le cas de [Gee95] et des travaux utilisant le logiciel GRAPHTALK [Niclet et al.96].

Il est d'ailleurs à remarquer qu'aucun travail sur l'étude d'un langage ne porte que sur la syntaxe concrète. Se cantonner à cette étude reviendrait à étudier les possibilités d'utilisation de signes sans savoir à quoi ils correspondent.

L'étude de la syntaxe abstraite

Étudier la syntaxe abstraite consiste à étudier les signifiés, les liens entre signifiés et les contraintes (sur ces signifiés et liens) pour que leur instanciation représente un modèle. Les langages utilisés pour la modélisation des SAP sont principalement des langages graphiques : langages de Merise, SADT, RdP, grafcet, etc. Certains travaux sur la méta-modélisation [Gee95] [Pierra et al.95] ont mis en évidence que ces langages peuvent être envisagés comme des structures particulières de graphes, orientés ou non, ou d'hypergraphes. A partir de ces structures, trois types de liens structurels entre signifiés sont à envisager :

- les liens non orientés d'association entre signifiés.
  Par exemple, dans le langage entité/relation [Chen76], le lien entre une entité et une relation n'est pas orienté.

- les liens orientés d'association entre signifiés.
  Par exemple, dans les différents types de réseaux de Petri [David et al.89], le lien entre une place et une transition est orienté.

- les liens de composition entre signifiés.
  Par exemple, dans le langage Statecharts [Harel87] le lien entre états est un lien de composition, de même dans SADT [Lissandre90] (actigramme ou datagramme). Dans le Grafcet [Bouteille et al.95], le lien entre une macro-étape et son expansion est aussi un lien de composition.

Les langages non graphiques, donc textuels, peuvent également être étudiés à travers ces différents liens.

Par exemple, dans le langage Express [Bouazza95] :

- certains opérateurs arithmétiques binaires représentent des liens non orientés entre signifiés (les opérandes),
- le lien de sous-typage entre entités est un lien orienté d'association entre signifiés,
- le lien entre un schéma et ses entités est un lien de composition entre signifiés.

Ces trois types de liens représentent donc l'ensemble des liens structurels possibles au sein d'un langage, qu'il soit graphique ou textuel. Étudier la syntaxe abstraite d'un langage nécessitera donc d'identifier les signifiés, les liens entre signifiés et les contraintes d'utilisation. Ces contraintes peuvent être des contraintes sur le nombre d'instance d'un signifié, d'un lien entre signifiés ou sur l'existence conjointe d'instances de différents signifiés ou liens.

Les deux étapes de l'étude de la syntaxe

L'étude de la syntaxe concrète est importante puisque les signes forment la partie visible du langage. Cependant dans les langages étudiés, les liens entre signes et signifiés ne sont pas ambigus. L'étude de la syntaxe pourra donc se faire en deux étapes. Une première étape consistera à étudier la syntaxe abstraite afin d'étudier les signifiés et les liens structurels du langage. Cette étape permettra également d'étudier l'intégration des langages au sein des méthodes. Une deuxième étape consistera à superposer à l'étude de la syntaxe abstraite, l'étude de la syntaxe concrète. Pour un nouveau langage, cette étude permettra de choisir des signes permettant de passer facilement des signes aux signifiés. Ce choix consistera par exemple à donner à chaque signe une forme différente (carré, cercle ...). La syntaxe concrète pourra en outre être enrichie lors de l'étude des méthodes de construction du modèle. Par exemple, les signes pourront changer de couleur, en fonction du respect de certaines contraintes, au cours de la construction du modèle, cela afin d'aider le concepteur à construire le modèle.

2.2.2 L'étude de la sémantique

La notion de sémantique est une notion qui est intuitivement très familière :

Sémantique [Gallimard92] : étude d'une langue ou de langues considérées du point de vue de la signification.

D'après notre définition du symbole, et les remarques précédentes sur l'utilisation des signes, nous considérons que l'étude de la signification ne comporte pas l'étude des signes.

Les signifiants des langages ne sont définis qu'à partir des signifiés et de leur liens non structurels. Nous proposons donc la définition :

Sémantique :	*ensemble des signifiants d'un langage. Ceux-ci sont définis à partir des signifiés et de leurs liens non structurels*

Le problème revient alors à savoir comment définir les signifiants des symboles?

Pascal Lhoste [Lhoste94] répond à cette question en proposant une distinction entre sémantique interne et sémantique externe (figure 2.6). La sémantique interne représente la définition des liens, non structurels, des différents signifiés d'un langage, alors que la sémantique externe représente l'ensemble des liens entre signifiés de différents modèles ou entre les signifiés d'un modèle et le système lui-même. Cependant dans les différents travaux présentés, l'étude de la sémantique interne est souvent réduite à l'étude de la dynamique des modèles [Lhoste94] [Bon bierel94].

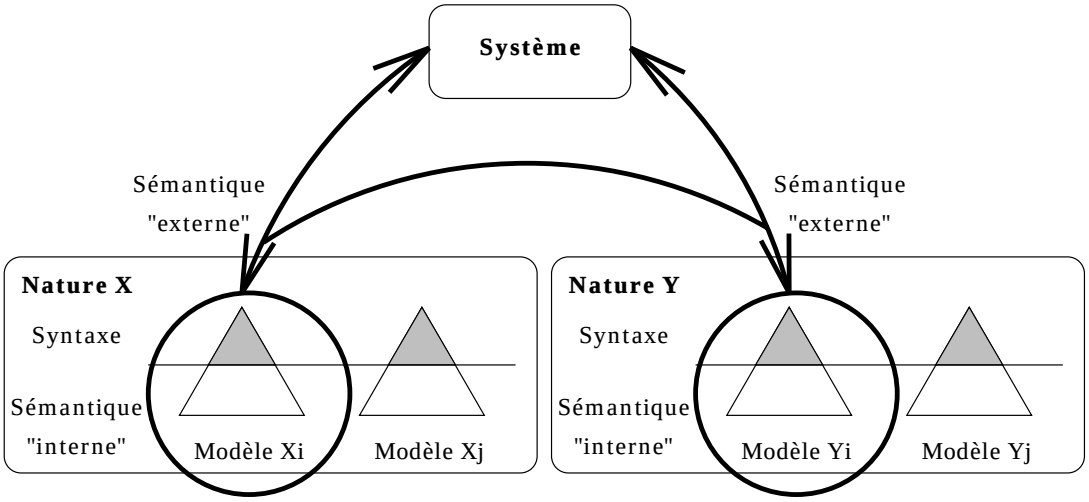


FIG. 2.6: *Sémantique interne et sémantique externe [Lhoste94]*

Ces définitions permettent de bien faire la distinction entre ce qui est interne au langage et ce qui dépend de son utilisation pour un système donné. La mise en relation de signifiés de différents langages, par contre, nous parait plus dépendre de l'aspect méthode qui sera examiné plus tard. Ceci restreint les définitions :

Sémantique interne :	***ensemble des signifiants internes d'un langage. Ces signifiants sont définis à partir des signifiés, de leurs liens non structurels et par rapport au temps.***
Sémantique externe :	***ensemble des signifiants externes au langage. Ces signifiants sont définis par rapport aux systèmes modélisés***

L'étude de la sémantique interne

Le premier aspect de l'étude de la sémantique interne est l'étude des liens non structurels (syntaxe abstraite) entre signifiés du langage. Donner une désignation à ces liens revient non seulement à donner un sens à ces liens, mais aussi par contrecoups un sens aux signifiés impliqués dans ce lien. Cette partie de la sémantique interne est finalement très proche de la syntaxe abstraite. La distinction entre les deux liens ne repose que sur la désignation des liens entre signifiés. Si le lien peut être considéré comme un lien structurel, il concerne la syntaxe abstraite, sinon il s'agit d'un lien sémantique.

L'autre aspect de la sémantique interne est l'aspect temporel. Les systèmes automatisés de production sont des systèmes dynamiques. Pour modéliser cette dynamique, certains langages prennent en compte le temps (au sens physique du terme) (SIGNAL [LP94]), d'autres utilisent la notion d'états successifs (réseaux de Petri [Proth et al.96]), d'autres enfin associent ces deux approches (réseaux de Petri P-temporisés [David et al.89]). Nous verrons plus tard que cette partie de l'étude de la sémantique interne est fortement liée à l'étude des méthodes de simulation des modèles produits.

L'étude de la sémantique externe

Donner un sens à la désignation des symboles utilisés dans un langage, c'est d'abord, et avant tout, leur donner un sens par rapport aux systèmes modélisés avec ce langage. Un actigramme SADT, un modèle conceptuel de données (MCD) de Merise ... doivent comporter un dictionnaire pour être compréhensibles. Ce dictionnaire apporte une sémantique externe au modèle, mais il a l'inconvénient de l'utilisation libre du texte : son manque de rigueur et ses problèmes d'interprétation. Une autre approche consiste à introduire dans le méta-modèle d'un langage des liens entre signifiés du langage et éléments d'un modèle des systèmes modélisables, les SAP. Ces éléments peuvent être considérés comme des signifiés dont les signifiants sont définis par des liens sémantiques. Par contre, ils ne correspondent pas à des symboles puisqu'ils ne sont pas associés à des signes. Nous parlerons alors de concepts - conjonctions d'un signifié et d'un signifiant - structurés éventuellement au sein d'une théorie :

Concept:	***conjonction d'un signifié et d'un signifiant.***

Théorie:	***ensemble de concepts et de leurs liens sémantiques.***

Ces définitions nous permettent en outre de rester cohérents vis à vis de la définition du langage. En effet, lors de l'association d'un langage à une théorie, il peut arriver qu'un

symbole de ce langage corresponde directement à un concept de la théorie associée : même signifié et mêmes signifiants. Ces définitions sont bien homogènes.

Un modèle est l'instanciation d'un langage sur un cas concret. Donner un sens aux signifiés d'un langage ne suffit pas pour s'assurer que les modèles construits auront un sens. Il faut aussi s'assurer que les instances des signifiés du langage et les structures construites à partir de ces instances ont chacune un sens. Les signifiants du langage permettent de définir un ensemble de modèles constructibles. A coup sûr, les modèles non constructibles n'ont pas de sens vis à vis du système modélisé. Cependant, tous les modèles constructibles n'ont pas non plus forcément un sens vis à vis de ce système. Il est donc indispensable de définir soit :

- les instances possibles des signifiés,
- les instances impossibles des signifiés,
- les instances possibles des liens entre signifiés,
- les instances impossibles des liens entre signifiés.

Ces contraintes sont liées au SAP modélisé. Elles viennent s'ajouter aux contraintes syntaxiques, qui, elles, sont vérifiées quel que soit le système modélisé.

La place de l'étude de la sémantique

L'étude de la sémantique - qu'elle soit interne ou externe - est indépendante de la syntaxe concrète du langage. Par contre, elle est complémentaire de l'étude de la syntaxe abstraite. Il nous semble donc que la première étape doit consister à étudier la syntaxe abstraite et la sémantique - interne et externe - du langage. La deuxième étape consiste à étudier la syntaxe concrète.

2.2.3 Relations entre signes, signifiés et signifiants

Prenons l'exemple de l'écriture d'une expression booléenne : elle peut être décrite par une équation mathématique, du texte ou encore par un schéma à relais (figure 2.7). Pourtant cette expression a toujours les mêmes signifiants et les mêmes signifiés, mais des signes différents. Cet exemple montre bien qu'à un signifié ou à un signifiant il ne correspond pas un seul signe. Cependant cet exemple correspond en fait à l'utilisation de plusieurs langages pour exprimer une même expression booléenne. Associer plusieurs systèmes de signes à un même système de signifiés et de signifiants consiste à définir plusieurs langages. Il apparaît ainsi que l'étude d'une syntaxe abstraite et d'une sémantique peut

permettre par l'ajout de syntaxes concrètes de définir conjointement plusieurs langages. Ces langages pourront être utilisés indifféremment puisqu'ils auront la même sémantique.

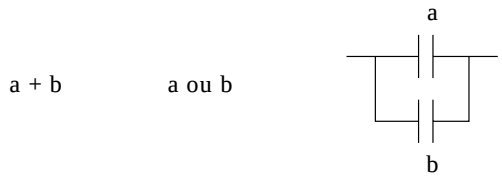


FIG. 2.7: *Différentes écritures d'une même expression booléenne*

Au sein d'un langage, nous considérons, par contre, qu'il est nécessaire que l'interprétation des signes soit univoque : c'est le déterminisme d'interprétation. Le lecteur d'un modèle, à la vue des signes qui constituent l'interface du modèle, doit pouvoir sans hésiter associer à chacun un seul signifié et signifiant. De même, il ne nous semble pas utile qu'à un signifiant et un signifié il puisse correspondre plusieurs signes au sein d'un même langage : c'est l'unicité de représentation. Cette unicité d'interprétation et de représentation nous conforte dans l'idée que l'étude de la syntaxe abstraite et l'étude de la sémantique sont les étapes les plus importantes dans l'étude d'un langage. L'étude de la syntaxe concrète est une étape nécessaire mais qui peut venir plus tard dans l'étude des langages.

L'unicité de représentation ne signifie d'ailleurs pas qu'il suffit de voir des signes pour en déduire le langage utilisé. Henri Habrias [Habrias93] a montré que la simple présentation de signes ne permet pas d'en déduire le signifié et le signifiant. En effet, beaucoup de langages utilisent les mêmes formes géométriques de base, cercle, triangle, rectangle ... pour exprimer des concepts totalement différents. Ceci confirme bien que l'étude de la syntaxe concrète, la « boxologie », ne doit pas être considérée comme le cœur de l'étude des langages.

2.3 L'étude d'une méthode

Les différents travaux portant sur l'étude d'une méthode concernent principalement l'étude de l'intégration de différents langages au sein d'une méthode. Dans un souci de clarté, notre exposé commencera par l'étude de la démarche de construction de modèles dans une méthode mono-modèle. En effet, les méthodes d'importation et d'exportation reprennent des éléments de ces méthodes mono-modèle en y ajoutant des opérations. Une fois que nous aurons vu comment construire des modèles, nous étudierons de quelles façons les valider puis les jouer.

Avant de commencer son étude détaillée, présentons rapidement la démarche de construction de modèles. Cette démarche utilise un langage pour produire un modèle. Cette mé-

thode est mono-modèle car nous considérons que la construction de plusieurs modèles correspond à :

- plusieurs méthodes de construction si les modèles sont indépendants ;
- une méthode intégrée si les modèles ne sont pas indépendants.

Par contre, notre représentation des opérations de construction et de vérification reste valable dans le cas de construction et de vérification d'un modèle construit à partir de plusieurs langages : elle correspond donc à des méthodes mono-modèle mais pas obligatoirement mono-langage. Ceci est également vrai pour les activités de validation et de jeu des modèles. Pourtant, à chaque fois que cela sera possible, nous ne représenterons qu'un seul langage afin de simplifier notre discours.

2.3.1 L'étude de la démarche de construction de modèles

Un langage est, avant toute chose, utilisé pour construire des modèles, et pourtant la démarche de construction de modèles est un sujet très peu développé. En effet, la plupart du temps, le concepteur est libre d'utiliser le langage comme il le souhaite pour construire un modèle. Pourquoi cette liberté ?

Construire un modèle est un travail difficile. C'est un travail de création dans lequel l'individu doit être libre de toute contrainte. Prenons l'exemple des actigrammes de SADT. Lors de la construction d'un modèle, l'auteur écrit des modèles intermédiaires qui peuvent contenir moins de trois activités par feuille, ou plus de six tant que son modèle n'est pas suffisamment hiérarchisé. Durant ces étapes de travail, les règles limitant le nombre d'activités par feuille n'est donc pas respecté. Par contre, à aucun moment, l'auteur ne devra pouvoir créer des symboles autres que des activités et des flux de données. Il en est ainsi de tout langage : à aucun moment un modèle ne peut contenir de symbole ne faisant pas partie du langage utilisé, par contre certaines règles contraignant l'utilisation des symboles peuvent être volontairement transgressées par l'auteur lors de certaines étapes de construction. Obliger le modélisateur à respecter tout de même certaines de ces règles rendrait impossible la construction d'un modèle ou la rendrait plus difficile. Une méthode de construction de modèles peut donc être appréhendée, du point de vue de l'utilisateur de la méthode, comme une association de deux opérations : une opération de construction et une opération de vérification du modèle. La figure 2.8 représente ces deux activités et leur intégration sous la forme d'un modèle fonctionnel. Sur cette figure, comme sur celles qui vont suivre, nous avons représenté les opérations sous forme d'activités, et leur intégration est vue à travers les données échangées entre elles.

Toute opération de construction peut être modélisée à partir de deux opérations élémentaires qui sont les opérations d'ajout et de suppression d'un symbole. Les opérations

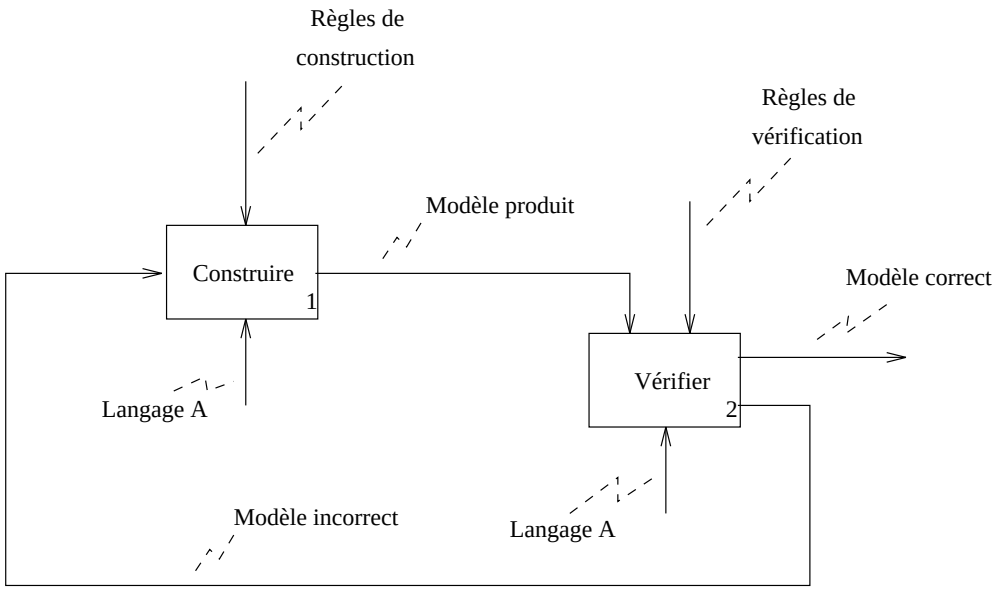


FIG. 2.8: *Les opérations d'une méthode de construction de modèles*

d'ajout ou de suppression de plusieurs symboles, de modification de symboles ... peuvent être vues comme la combinaison de plusieurs opérations d'ajout et de suppression d'un symbole. L'opération de construction nécessite l'utilisation d'au moins un langage et de règles permettant d'assurer la cohérence (syntaxique et sémantique) du modèle produit, sans pour autant limiter la créativité de l'auteur. L'opération de vérification consiste à vérifier que les règles limitant cette créativité, mais nécessaires à l'expression du modèle, soient vérifiées. Elle nécessite également l'utilisation du même langage.

La séparation entre les règles utilisées dans les opérations de construction et de vérification est arbitraire. Un bon compromis entre liberté d'utilisation des langages et rigueur des modèles obtenus nous semble être de respecter toutes les règles syntaxiques et sémantiques dans l'opération de construction, sauf les contraintes syntaxiques sur les instances, puis de vérifier ces règles dans l'opération de vérification. Cependant, si l'opération de construction se fait avec un simple éditeur, il sera nécessaire de vérifier beaucoup plus de règles dans l'opération de vérification.

Dans notre représentation de l'opération de construction, nous n'avons représenté aucune entrée, autre que celle produite par l'activité elle-même. En règle générale l'auteur dispose de modèles construits en amont du cycle de conception, et au moins d'un cahier des charges. On peut considérer, ou du moins espérer, que le concepteur dispose toujours au moins d'un document écrit, sauf s'il a lui-même la charge de la rédaction du cahier des charges. D'après nos définitions, ce travail correspond aussi à l'utilisation d'une méthode de construction d'un modèle.

2.3.2 L'étude de l'intégration de langages

François Kiefer a étudié l'intégration de langages dans [Kiefer96] . L'intégration y est vue comme une mise en relation des différents signifiés des deux langages intégrés. L'utilisation du langage Entité/Association permet de représenter les différents signifiés et leurs liens. Bien sûr les signifiés ne sont vus que sous leur aspect statique, les aspects temporels ne pouvant pas être pris en compte par le langage Entité/Association. La démarche utilisée, la quantité de langages méta-modélisés, ont permis à François Kiefer d'effectuer un travail exhaustif sur les liens sémantiques (correspondances statiques) entre signifiés de deux langages. Les différents cas de correspondance entre signifiés de deux langages sont :

- la correspondance directe entre signifiés : les deux langages comportent chacun un symbole ayant la même sémantique ;
- la correspondance indirecte à iso-sémantique : un des deux langages comporte plusieurs symboles ayant la même sémantique que le symbole considéré de l'autre langage, ceci correspond donc à deux cas possibles (réflexivité) ;
- la correspondance indirecte avec degrés de détail de sémantique différents : un des deux langages comporte plusieurs symboles dont la sémantique correspond à une partie de la sémantique d'un seul symbole de l'autre langage, ceci correspond aussi à deux cas possibles (réflexivité).

Au delà de ces cinq cas de correspondance de symboles entre langages, les travaux de François Kiefer avaient pour objectif l'étude de la construction de modèles au sein d'une méthode intégrée. Nous avons déjà dit que l'élaboration d'un modèle nécessite deux opérations : la construction et la vérification. Lorsque ce langage est intégré au sein d'une méthode, elle utilise deux autres opérations : l'importation et l'exportation (figure 2.9).

L'importation consiste à construire des symboles dans le modèle en cours de construction à partir de modèles existants. L'exportation, qui est l'opération inverse, peut avoir deux objectifs. Elle peut servir à enrichir des modèles existants : c'est notamment le cas lorsque l'intégration se fait par l'intermédiaire d'un langage pivot. Elle peut aussi être utilisée pour valider le modèle en cours de construction par rapport aux modèles précédents, en vérifiant justement que le résultat de l'exportation soit cohérent avec les modèles existants. Cette modélisation de l'intégration des langages au sein d'une méthode d'élaboration de modèles est valable aussi bien pour une intégration directe – c'est-à-dire l'intégration des langages utilisés au sein d'une même étape du cycle de vie ou d'étapes successives – que pour une intégration par pivot – les langages de la méthode sont intégrés à un seul langage, le pivot (figure 2.10). Si l'utilisation du langage pivot nécessite *a priori* moins de travail, il n'est pas évident de trouver un langage pivot dont les signifiés peuvent être mis

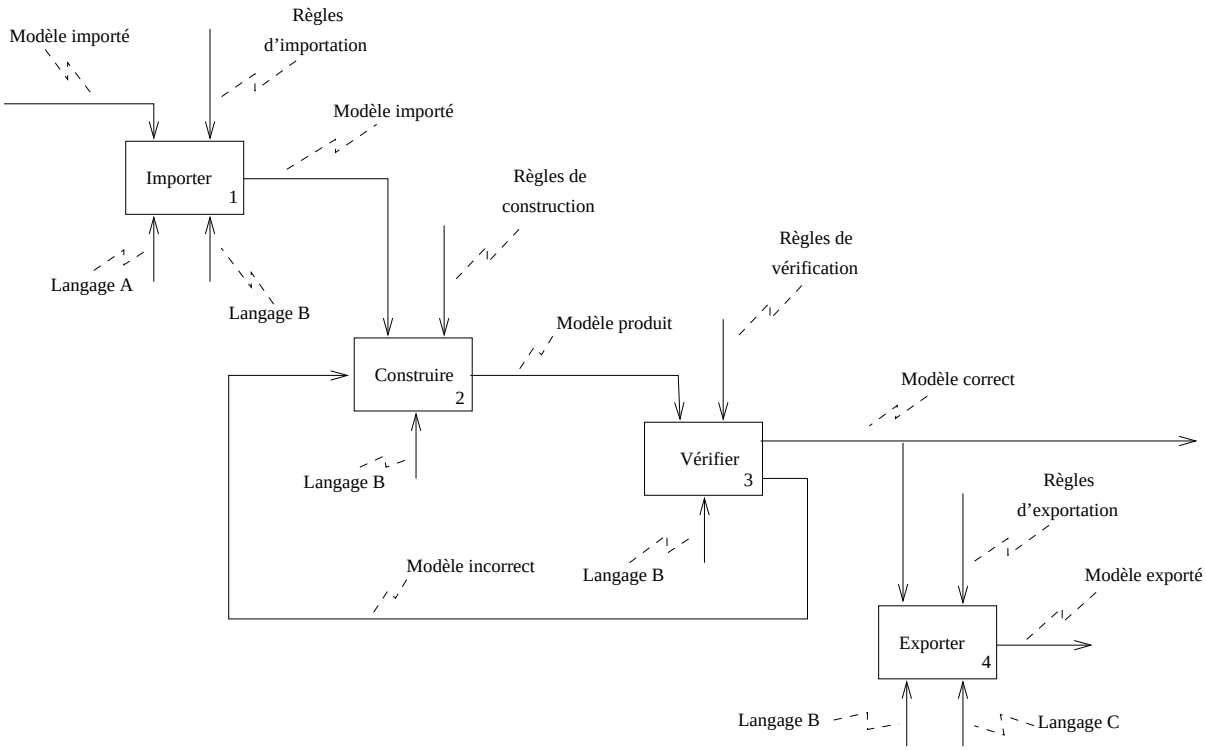


FIG. 2.9: *Les opérations d'une méthode intégrée de construction de modèles*

en relation avec tous les signifiés de tous les langages utilisés. Une méthode pourra donc contenir des langages intégrés par pivot, tandis que d'autres seront intégrés directement, en particulier dans les phases plus « linéaires » du cycle de conception du SAP.

Les règles à observer pour les opérations d'importation et d'exportation sont donc :

- des règles de choix d'un symbole parmi une liste de symboles possibles, dans le cas de correspondance indirecte à iso-sémantique ;
- des règles de vérification d'intersection de décomposition dans le cas de correspondance indirecte avec degrés de détail de sémantique différents.

2.3.3 L'étude de la validation

L'exemple de la validation du GRAFCET par automate équivalent [Roussel94] permet de présenter les besoins (figure 2.11) des opérations de validation. La démarche d'utilisation d'AGGLAE (logiciel d'Analyse de Grafcets par Génération Logique de l'Automate Équivalent) peut se décomposer en cinq activités. L'objectif de l'utilisation d'AGGLAE est la validation d'un modèle GRAFCET. Cette validation nécessite la construction d'un automate à états équivalent. Nous retrouvons là un exemple d'intégration entre deux langages. Cette intégration comporte ici trois opérations : les opérations de construction et

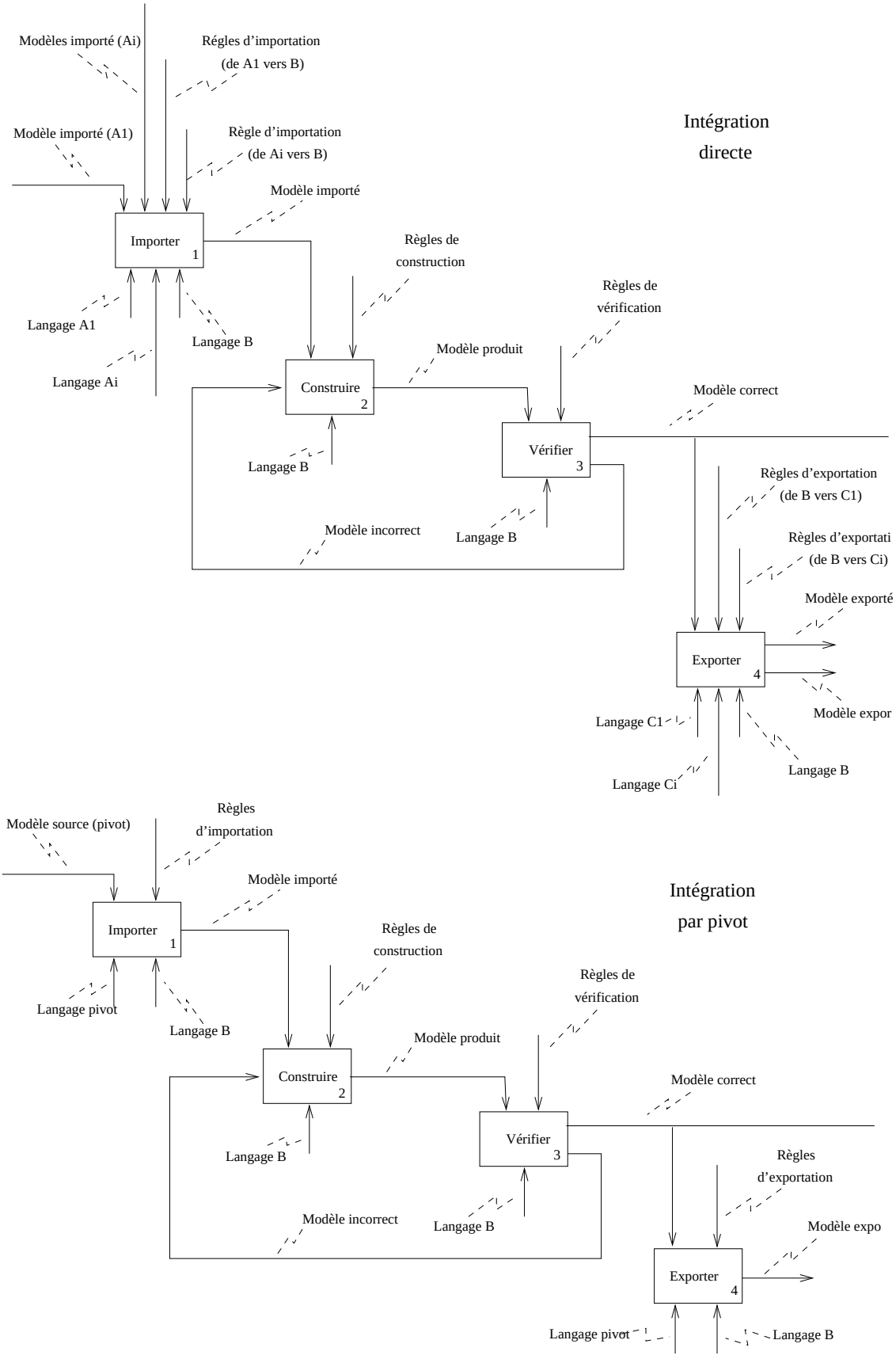


FIG. 2.10: *Intégrations multiples d'une méthode de construction de modèles*

de vérification du modèle GRAFCET (qui sont préliminaires à l'utilisation proprement dite d'AGGLAE), et l'opération d'exportation du modèle GRAFCET vers l'automate à états. Cette exportation est une exportation directe entre signifiés. Ces trois opérations nécessitent deux langages (le GRAFCET et les automates à états) et des règles de type règles de construction et règles de vérification que nous avons déjà étudiées.

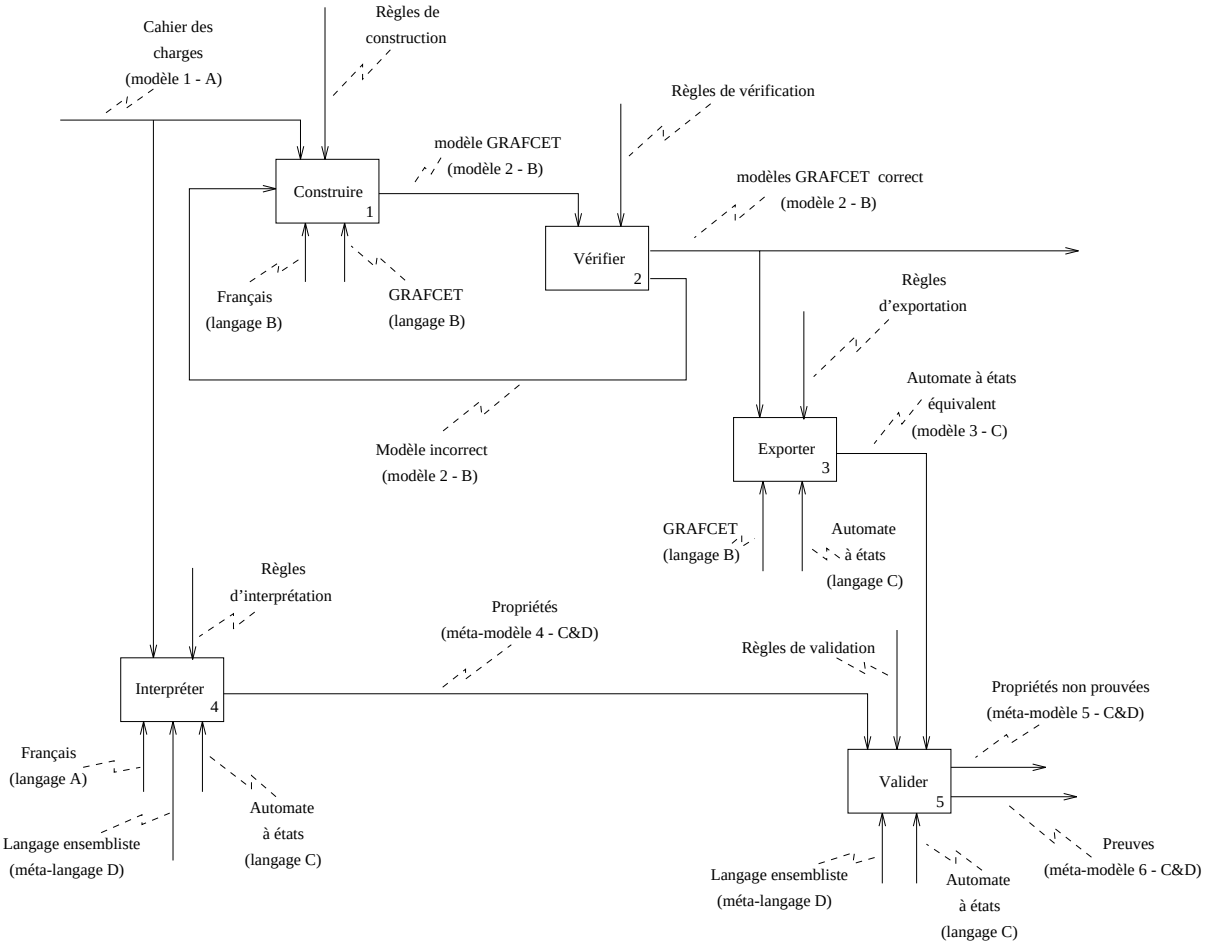


FIG. 2.11: *Validation de modèle GRAFCET par automate équivalent*

La dernière opération - l'activité de validation sur la figure 2.11- nécessite des règles de validation permettant de manipuler des portions du modèle pour prouver les propriétés recherchées. Il est nécessaire d'interpréter - dans l'activité 4 - le cahier des charges afin que ces propriétés soient exprimées dans le même langage que le modèle à valider. Cependant, ce langage - les automates à états - n'est pas suffisant à l'expression des propriétés. En effet, la preuve des propriétés nécessite la manipulation d'expressions issues du modèle: il est donc nécessaire de passer au niveau méta-modèle. Les automates à états ayant été définis à l'aide de la théorie des ensembles, les propriétés sont écrites à l'aide d'opérateurs ensemblistes manipulant des ensembles issus du modèle à valider. Les règles d'interpréta-

tion de cette opération ne sont pas écrites dans AGGLAE, et pour cause ! Ces règles font appel à la fois à de multiples connaissances :

- la connaissance du français, du GRAFCET et du langage ensembliste pour ce qui concerne les langages ;
- la connaissance approfondie des parties commande et opérative des SAP, et maîtrise du processus concerné en ce qui concerne le SAP conçu.

L'activité 5 - la validation - consiste à dérouler la preuve. Elle nécessite également l'utilisation du langage ensembliste et de la définition du langage des automates à états. Les règles à utiliser sont les règles habituelles d'utilisation des opérateurs de ce langage ensembliste. Ces règles ne sont pas non plus triviales à utiliser.

Dans le cas général, comme sur cet exemple du grafcet, l'opération de validation consiste à prouver qu'un modèle a bien certaines propriétés tirées du cahier des charges. L'expression de ces propriétés nécessite l'utilisation d'un deuxième langage permettant de manipuler des expressions, ces expressions étant elles-mêmes exprimées dans le langage utilisé pour construire le modèle à valider. Ce deuxième langage est donc utilisé en tant que méta-langage. Les propriétés exprimées ainsi que les preuves effectuées sont donc des méta-modèles.

2.3.4 L'étude du comportement dynamique

Pour la conception des SAP, les modèles exécutables sont généralement construits à partir de langages à états. Les différentes classes de RdP et le GRAFCET sont les langages les plus couramment utilisés. La dynamique de ces langages n'est généralement exprimée que sous forme de règles de transformations ou d'équations conditionnelles ([Murata89] par exemple). D'autres travaux expriment les conditions d'utilisation de ces règles sous la forme d'un algorithme de jeu [Lhoste et al.97]. Les règles associées à l'opération de jeu sont donc des règles exprimant la succession, conditionnelle ou non, ou la simultanéité d'opérations élémentaires. Ces opérations élémentaires permettent de déterminer des propriétés du modèle ou d'en modifier l'état.

2.3.5 La place de l'étude d'une méthode

Nous avons mis en évidence sept types d'opérations associées à des langages : les méthodes de construction, de vérification, d'importation, d'exportation, de validation, d'interprétation et de jeu de modèles. Les opérations de vérification et de validation ne modifient pas les modèles. Elles peuvent donc, *a priori*, être étudiées alors que les langages

utilisés sont déjà définis (à part la syntaxe concrète bien sûr). Par contre, toutes les opérations d'importation, d'exportation, de construction et de jeu modifient le modèle. Pour les méthodes intégrant ces opérations, il est donc nécessaire de les étudier en même temps que la syntaxe abstraite et la sémantique des langages qu'elles utilisent. Il peut arriver qu'une nouvelle méthode soit définie à partir de langages existants. Dans ce cas, la définition de cette méthode risque d'entraîner des modifications de la définition des langages utilisés. Ceci crée des distorsions entre les méthodes utilisant un langage qui devrait être unique alors que sa définition ne l'est pas. C'est ce problème qui explique, en partie, que certains langages ont de nombreuses variantes dans les méthodes utilisées pour des besoins particuliers. Ce problème n'a *a priori* pas de solution : c'est d'ailleurs ce qui fait tout l'intérêt de la méta-modélisation.

2.4 Les méta-modèles

2.4.1 Niveau objet et niveau méta

Dans le chapitre 1, nous avons montré que les approches existantes en méta-modélisation consistent à décrire la structure des modèles de SAP. Pourtant, dans les parties précédentes de ce chapitre, nous avons étudié tous les aspects des langages et des méthodes, et déclaré qu'ils doivent être étudiés dans les méta-modèles pour définir complètement l'activité de conception d'un SAP. Ceci nous incite donc à préciser nos définitions de modèle et de méta-modèle. Dans le cadre de nos travaux, l'objet de modélisation est le SAP. Par abus de langage, la définition de modèle devient :

Modèle :	**Instanciation d'un langage pour la modélisation d'un SAP ou d'une famille de SAP.**

Quant à la définition de méta-modèle :

Méta-modèle :	**Instanciation d'un langage pour la modélisation de l'activité de modélisation d'un SAP.**

[Courtier et al.93] définit trois niveaux de représentation du réel : le niveau application, le niveau modèle et le niveau méta-modèle (figure 2.12). La distinction entre le niveau application et le niveau modèle ne nous parait pas fondée. En effet, dans ces deux niveaux les modèles représentent le SAP. Bien sûr, le niveau modèle est plus générique que le niveau application. Cependant, il peut exister de nombreux niveaux de généralisation lorsque le concepteur choisit de modéliser des familles de systèmes. Il nous parait donc légitime de ne distinguer que deux niveaux de modélisation. En linguistique, il existe deux

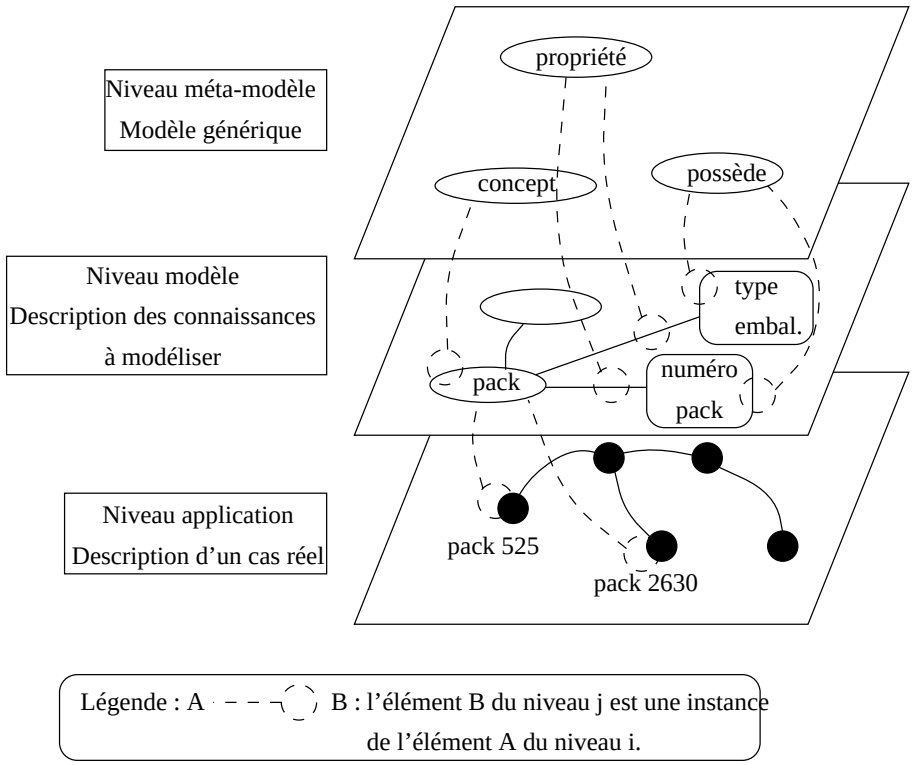


FIG. 2.12: *Les trois niveaux de la représentation selon [Courtier et al.93]*

niveaux d'utilisation du langage: le niveau objet et le niveau méta. Ces deux niveaux correspondent bien à notre approche et sont suffisamment génériques pour être adoptés, bien que le terme objet ait un sens particulier en informatique. Dans notre cas, le niveau objet correspond à l'utilisation de langages et de méthodes pour produire des modèles de SAP. Nous parlerons alors de langages, méthodes et modèles. Le niveau méta correspond à l'utilisation de langages et de méthodes pour produire des modèles des langages et des méthodes utilisés pour la modélisation des SAP. Nous parlerons alors de méta-langages, méta-méthodes et méta-modèles (figure 2.13).

2.4.2 Deux niveaux suffisent-ils ?

Après avoir défini un niveau objet et un niveau méta, il est légitime de se demander s'il est nécessaire de définir d'autres niveaux.

Un troisième niveau peut consister à modéliser les langages et méthodes utilisés pour méta-modéliser (un niveau méta-méta). Ce niveau permet alors de mieux spécifier les langages utilisés. Il permet surtout de formaliser les méthodes utilisées pour méta-modéliser. Nous remarquons alors qu'au niveau méta comme au niveau méta-méta, les systèmes modélisés sont des langages et des méthodes (figure 2.14). Il est donc probable que les

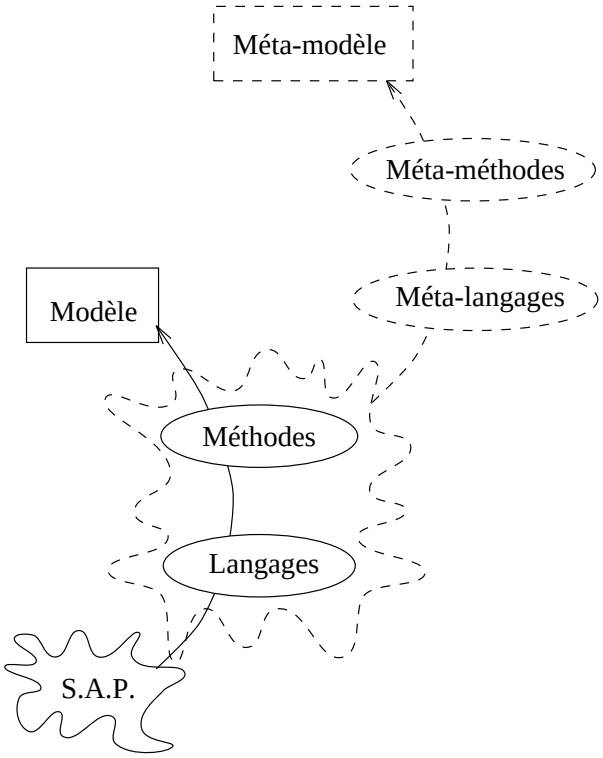


FIG. 2.13: *Les deux niveaux de modélisation*

langages et méthodes utilisés dans ces deux niveaux soient les mêmes. Les modèles produits au niveau méta-méta sont donc récursifs puisqu'ils se modélisent eux-mêmes ! Un quatrième niveau (méta-méta-méta !) n'est donc pas nécessaire puisqu'il serait identique au niveau méta-méta. Ajoutons tout de même que ce niveau méta-méta n'est pas du domaine du spécialiste en conception des SAP, mais uniquement du domaine du spécialiste en méta-modélisation.

2.4.3 Des besoins complémentaires

Jusqu'à présent, nous avons défini les différentes notions qui doivent être méta-modélisées pour rendre plus rigoureuse l'activité de modélisation d'un SAP. Sans vouloir détailler davantage la description des méthodes de méta-modélisation, deux caractéristiques de ces méthodes nous paraissent présenter un intérêt particulier : la vérification et la validation des méta-modèles.

Nous l'avons déjà dit, la qualité des modèles produits dépend de la qualité des méta-modèles. Il nous semble donc très important de pouvoir vérifier ces méta-modèles, c'est-à-dire vérifier que les règles syntaxiques et sémantiques des méta-méta-langages et méta-méta-méthodes sont bien respectées. Les méta-méta-langages et méta-méta-méthodes utilisés pour méta-modéliser doivent donc permettre de prouver ces propriétés de cohérence

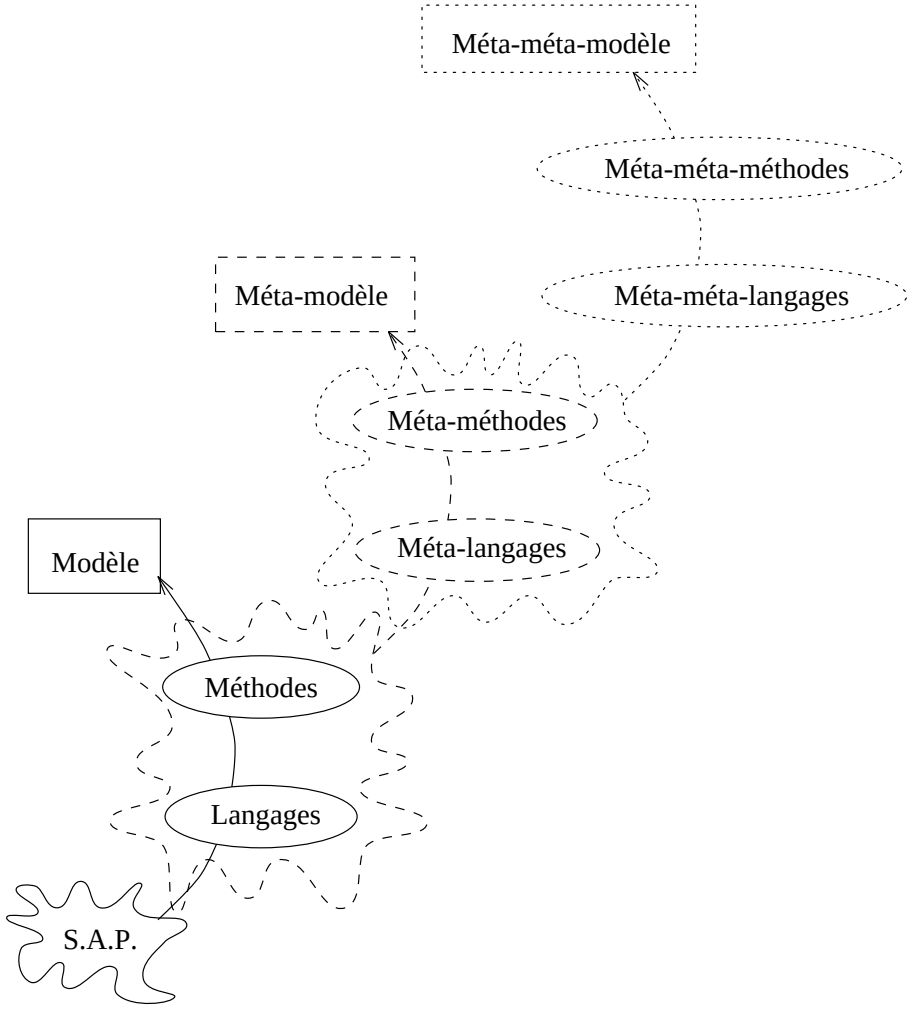


FIG. 2.14: *Les trois niveaux de modélisation*

syntaxique et sémantique.

En outre, si méta-modéliser un langage et une méthode produit toujours un résultat, encore faut-il que ce résultat corresponde aux attentes des utilisateurs. Il est donc nécessaire de vérifier que ces méta-modèles correspondent bien aux langages et méthodes souhaités par ces utilisateurs. Il est donc nécessaire de valider ces méta-modèles par rapport à ces besoins. Cette validation peut se faire de deux manières. Tout d'abord, les utilisateurs peuvent définir un certain nombre d'instanciations des méthodes et langages souhaités. Les méta-modèles doivent donc également pouvoir être instanciés pour obtenir ces résultats. De plus, certaines propriétés des langages et des méthodes peuvent être énoncées et vérifiées à partir des méta-modèles. Elles sont alors exprimées dans le méta-méta-langage. Ces propriétés doivent donc pouvoir être prouvées à l'aide du méta-méta-langage.

Enfin, un dernier point ne peut être négligé: l'élaboration de logiciels d'exploitation des langages et méthodes de conception des SAP. Jusqu'à présent, nous avons considéré

qu'un méta-modèle était une fin en soi. Pourtant, dans le cadre d'utilisation de logiciels pour modéliser, cela ne peut pas suffire. En effet, la production de logiciels servant à utiliser des langages et méthodes ne peut pas être indépendante de la définition par méta-modélisation de ces langages et méthodes. Il serait donc souhaitable que la production de ces logiciels se fasse à partir de ces méta-modèles, ou, au moins, en vérifiant que la structure des données de ces logiciels correspond bien à la structure des symboles des méta-modèles.

2.5 Récapitulatif des attentes en méta-modélisation

Les attentes en méta-modélisation sont donc très importantes. Les différents aspects des langages à méta-modéliser sont :

1. la syntaxe abstraite :
   - les signifiés,
   - les liens structurels non orientés d'association entre signifiés,
   - les liens structurels orientés d'association entre signifiés,
   - les liens structurels de composition entre signifiés,
   - les contraintes d'utilisation des signifiés et des liens ;
2. la syntaxe concrète :
   - les signes,
   - les liens entre signes et signifiés ;
3. la sémantique interne :
   - liens sémantiques entre signifiés du langage,
   - prise en compte du temps (physique ou séquentialité) ;
4. la sémantique externe :
   - les concepts de la théorie (signifiés et signifiants),
   - les liens sémantiques entre signifiés du langage et de la théorie,
   - les instances possibles et impossibles des signifiés et des liens entre signifiés.

Les autres aspects des méthodes à méta-modéliser sont :

1. la démarche d'intégration des opérations,

2. les opérations :
   - de construction,
   - de vérification,
   - d'importation,
   - d'exportation,
   - de validation,
   - d'interprétation,
   - de jeu ;
3. les règles associées à ces opérations.

En outre, le méta-méta-langage doit permettre la vérification et la validation des méta-modèles.

En définissant toutes ces attentes nous avons posé des bases solides pour tout travail de méta-modélisation. Cette liste nous permet ainsi de vérifier que les travaux de méta-modélisation existants sont loin de couvrir l'ensemble des besoins. Dans le chapitre trois, nous allons proposer notre approche de méta-modélisation dont l'objectif est de prendre en compte l'ensemble des attentes. Après une comparaison rapide des langages formels, nous présenterons les éléments du langage utilisé comme méta-méta-langage : le langage formel Z. Le plan de la suite du chapitre 3 reprendra successivement chacune des attentes exprimées ici pour développer les éléments de solution correspondants à ces attentes.

Chapitre 3

Solution apportée aux besoins

Le chapitre précédent nous a permis de définir l'ensemble des besoins en méta-modélisation. Nous avons détaillé les besoins de méta-modélisation des langages et ceux de méta-modélisation des méthodes. Nous avons également montré l'importance de deux besoins supplémentaires : la vérification et la validation des méta-modèles. L'ensemble de ces besoins nous a amené à nous tourner vers les langages formels et parmi ceux-ci à rechercher un langage permettant de modéliser la structure des données et les opérations sur celles-ci.

Ce chapitre commencera donc par une présentation rapide de quelques langages et méthodes formels qui nous permettra de justifier notre choix du langage Z. Nous consacrerons ensuite quelques pages à la description des différents aspects du langage Z. Cette présentation sera une présentation des principes, le lecteur désirant plus de détails pourra se rapporter à l'annexe A de ce document. Après cette présentation, nous reviendrons alors sur chacun des besoins énumérés au chapitre précédent pour présenter en détail les solutions envisagées avec le langage Z. Lorsque cela sera possible, nous présenterons ces solutions sur des exemples courts. Nous pourrons alors conclure sur les possibilités de Z en méta-modélisation.

3.1 Les langages formels

Il existe de nombreuses méthodes formelles utilisées pour des applications très différentes. [Bowen et al.93] présente un ensemble très important d'applications industrielles des méthodes formelles en Grande-Bretagne, pays où ces méthodes sont très utilisées. Confronté au choix d'une méthode formelle, nous nous sommes intéressés aux méthodes les plus connues et les plus diffusées. Parmi les méthodes formelles, les plus connues et les mieux documentées sont les types abstraits algébriques, LOTOS, VDM et Z [Gaudel et al.96].

Les spécifications algébriques [Monin96] sont déclaratives, ce qui leur donne un carac-

tère abstrait. Elles sont structurées et modulaires. Les logiques associées sont simples, et il existe des outils pour assister les preuves. Ces spécifications sont bien adaptées à la définition des données et à la description statique des systèmes informatiques. Par contre, elles sont moins adaptées aux aspects dynamiques. Les méthodes de spécifications algébriques ne seraient donc pas satisfaisantes pour notre étude car elles ne permettraient pas l'étude de la construction des modèles et leur jeu.

Le langage LOTOS permet la spécification de systèmes répartis où des actions s'exécutent en parallèle et doivent parfois se synchroniser. Ce langage est utilisé dans le domaine des télécommunications pour décrire et valider des protocoles de communication. La structuration des données n'est pas du domaine d'application de LOTOS qui ne correspond donc pas à nos attentes de méta-modélisation.

VDM [Jones90] [Aristide90] et Z [Spivey94] [Lightfoot94] sont orientés vers la description de systèmes avec états. La spécification des états possibles est effectuée en partant de types de données prédéfinis. L'écriture d'une spécification dans ces langages passe d'abord par la définition de l'état manipulé ; elle se poursuit par la spécification des opérations manipulant cet état. Une des différences entre ces deux langages est l'utilisation d'une logique à trois valeurs pour VDM et à deux valeurs pour Z. Grâce à une bonne modélisation de la structure des données et des opérations sur ces données, ces deux langages semblent correspondre à nos besoins. Le langage Z a connu un développement et une diffusion beaucoup plus importante que VDM et fait l'objet d'une littérature conséquente. Ces raisons nous ont poussé à choisir Z plutôt que VDM, et à tester notre choix sur un exemple de méta-modélisation d'un langage de modélisation des SAP [Pietrac et al.96].

Comme Z, la méthode B a été développée par J.-R. Abrial [Abrial96]. Bien que basée aussi sur la théorie des ensembles et la logique des prédicats, sa syntaxe et sa sémantique sont très différentes du langage Z. L'élément de structuration est la machine abstraite qui encapsule les déclarations et les opérations sur les éléments déclarés (l'exemple suivant présente le raffinement d'une machine abstraite spécifiant le comportement d'un carrefour avec deux feux tricolores [Chauvet96]). B est une méthode car, outre l'étape de spécification, elle inclut des étapes de raffinement qui permettent d'aboutir à du code (Ada ou C par exemple). Des mécanismes de preuve permettent de vérifier et de valider les modèles. La méthode B est supportée par deux logiciels commerciaux : B-Toolkit en Grande-Bretagne, et Atelier-B en France. La méthode B ne bénéficie pas de la même activité scientifique que Z (propositions d'extensions, développement de logiciels de preuves, de simulation ...), c'est pourquoi nous avons préféré choisir Z et non B.

REFINEMENT CARREFOUR1
REFINES CARREFOUR

VARIABLES
feu1, feu2

INVARIANT
$feu1 = feuA \wedge$
$feu2 = feuB$
INITIALIZATION
feu1, feu2 := jaune, jaune

OPERATIONS
MiseEnService $\widehat{=}$
BEGIN
feu1, feu2 := rouge, vert
END;

ChangerFeux $\widehat{=}$
BEGIN
$feu2 = vert$ THEN
feu2 := jaune
ELSE
feu1, feu2 := suiv(feu1), suiv(feu2)
END

FIG. 3.1: *Exemple de machine abstraite [Chauvet96]*

3.2 Le langage Z

3.2.1 Les ensembles

Le langage Z est basé sur une théorie des ensembles. La théorie dite naïve des ensembles (ou théorie de Cantor) est sujette à quelques paradoxes dont le plus connu est celui de Russel. En effet cette théorie permet d'exprimer un ensemble R tel que $R = \{x \mid \neg\ (x \in x)\}$. Plus récente, l'axiomatique de Zermelo-Fraenkel introduit une stratification qui permet de distinguer les individus, les ensembles d'individus, les ensembles d'ensembles, etc. Cette stratification permet de supprimer les paradoxes de la théorie de Cantor. Le langage Z utilise en plus la notion de typage (un type est un ensemble dans Z). Ceci permet de poser des limites aux opérateurs applicables sur les ensembles, permettant ainsi de vérifier leur bon usage.

Le premier chapitre a montré que l'approche la plus répandue et la plus avancée de méta-modélisation est basée sur des modèles de données, le plus souvent construits avec un langage de type entité/relation. Cependant, les notions d'entité et de relation définies dans ce langage ne sont pas toujours simples à distinguer. Les notions d'ensembles, d'individus et de relations par contre sont sans ambiguïté. Elles nous ont donc paru plus faciles à

utiliser pour méta-modéliser, d'autant que la théorie des ensembles dispose d'opérateurs permettant la construction d'ensembles à partir d'autres ensembles.

Dans le langage Z, tous les individus et les ensembles sont typés. Un individu ou un ensemble n'a qu'un type, et il n'est pas possible de convertir un élément (le terme élément sera employé à la place de « individu ou ensemble » pour simplifier notre discours) d'un type en élément d'un autre type. Un type n'a pas de sous-type. Un type doit être déclaré avant d'être utilisé dans la définition d'un élément. A part cette définition de type, l'utilisation des ensembles et individus dans le langage Z est identique à l'utilisation « courante » des ensembles. Nous pourrons ainsi utiliser les opérateurs classiques ($\in$, $\cup$, $\cap$...), la définition d'ensembles en compréhension ($A = \{x : X \mid x \notin B\}$) ou en extension ($A = \{paul, jacques\}$).

3.2.2 La logique

Une des limites courante des méta-modèles exprimés dans un langage graphique de modèles de données est la nécessité d'ajouter des contraintes textuelles faisant partie du modèle. Les opérateurs de la théorie des ensembles et les opérateurs logiques permettent *a priori* d'exprimer toute contrainte. La cohérence entre la modélisation des données et celle des contraintes associées ne se pose pas dans notre cas puisque les deux seront exprimées dans le même langage. La théorie des prédicats du premier ordre permet de définir des ensembles, des relations, des fonctions contraintes. Elle ne permet pas, à elle seule, d'exprimer des fonctions de fonctions (logique du deuxième ordre), des fonctions de fonctions de fonctions (logique du troisième ordre), etc. Cependant, ces fonctions de fonctions de fonctions ... sont exprimables grâce à l'association de la logique des prédicats du premier ordre et d'une théorie axiomatique des ensembles.

Là encore, l'utilisation de la logique des prédicats du premier ordre dans le langage Z est très proche de l'utilisation « courante » de cette logique. Nous y retrouvons les opérateurs de la logique des propositions ($\neg$ (non), $\wedge$ (et), $\vee$ (ou), $\Rightarrow$ (implique) et $\Leftrightarrow$ (équivaut à)), les quantificateurs ($\forall$ (quel que soit), $\exists$ (il existe)) et l'expression des prédicats, dans lesquels cependant les éléments déclarés doivent être typés ($\forall x : X \bullet p$ (quel que soit x de type X, alors p)).

3.2.3 Les éléments de structuration

Le langage Z dispose d'un élément de structuration des spécifications : le schéma. Un schéma permet d'encapsuler des éléments de spécification tels que les définitions d'individus ou d'ensembles, ainsi que les prédicats appliqués à ces définitions.

Exemple de schéma

Partie déclarative

Partie prédicative

La notion d'état du système spécifié permet d'utiliser un schéma pour spécifier soit cet état, soit une opération sur cet état, rendant ainsi la spécification très lisible. Les deux états des variables, avant et après l'opération, sont distingués par l'utilisation d'une apostrophe. *Variable* et *Variable'* représentent donc le même élément, avant (*Variable*) et après (*Variable'*) l'opération. Les définitions des variables sont locales (internes au schéma) au schéma Z. Il faut donc déclarer les variables de l'état dans les schémas d'opérations. Cela revient à déclarer les deux états du schéma définissant l'état du système spécifié. Si l'opération ne modifie pas cet état cela revient à déclarer Ξ*État*, soit *État* et *État'* avec *État'* = *État*.

État

Partie déclarative

Partie prédicative

Opération1

Ξ*État*

Déclaration des entrées et sorties

Prédicats spécifiant l'état des variables après opération

Si l'opération modifie cet état cela revient à déclarer Δ*État*, soit *État* et *État'* avec *État'* $\neq$ *État*.

Opération2

Δ*État*

Déclaration des entrées et sorties

Prédicats spécifiant l'état des variables après opération

3.2.4 Les preuves

Le langage Z utilise de nombreux symboles qui ne se trouvent pas sur un clavier. Écrire des modèles en langage Z nécessiterait donc, avec la plupart des traitements de textes, de dessiner ces symboles. Heureusement LaTeX dispose de styles prédéfinis permettant d'écrire ces symboles : zed.sty pour LaTeX2.09, oz.sty et z-eves.sty pour LaTeX 2_ε.

Quelques logiciels gratuits permettent en outre de faire de la vérification de type ou certaines preuves sur une spécification. Ils permettent juste de s'assurer que de « grosses bêtises » n'ont pas été écrites, mais c'est déjà beaucoup, et en fait tout à fait indispensable. Nous avons ainsi utilisé ZTC [Jia95a], un vérificateur de type, ZANS [Jia95b], un simulateur (pour les spécifications très simples) qui inclut une version récente de ZTC, et Z-EVES [Meisels et al.97] qui permet de faire quelques preuves (par exemple prouver que l'état initial existe bien).

Dans la suite de ce chapitre, le plan est structuré conformément aux attentes définies dans le chapitre précédent, et rappelées dans la section 2.5.

3.3 Méta-modélisation des langages

3.3.1 La syntaxe abstraite

Les signifiés

Comme nous l'avons sous-entendu précédemment, le langage Z permet de modéliser les signifiés par des individus ou des ensembles typés.

Une première utilisation consiste à définir les signifiés uniquement par leur nom, ce qui revient à déclarer des types de base. Par exemple, pour le langage grafcet, nous pouvons définir les types *ÉTAPE* et *TRANSITION* [Pietrac et al.96]. A partir de ces types, nous pouvons aussi bien définir des individus (a : *ÉTAPE*), que des ensembles finis ($b : \mathbb{F}\, \textit{ÉTAPE}$) ou infinis ($c : \mathbb{P}\, \textit{ÉTAPE}$), éventuellement non vides ($d : \mathbb{F}_1\, \textit{ÉTAPE}$ ou $e : \mathbb{P}_1\, \textit{ÉTAPE}$). La dénomination de ces éléments (individus ou ensembles) est quelconque, mises à part les contraintes d'utilisation des décorations (« ' », « ! », « ? ») et des nombres (par exemple « 2 » fait partie de $\mathbb{N}$ et ne peut pas être déclaré élément de *ÉTAPE*). Dans une spécification, la déclaration de ces signifiés apparaît à trois endroits :

- tout d'abord dans la déclaration des types :

$$[TYPE1, TYPE2]$$

- dans la déclaration des individus et ensembles de l'état du système spécifié :

Etat

$individu1 : TYPE1$

$ensemble1 : TYPE1$

$ensemble2 : TYPE2$

...

– dans la déclaration des entrées et sorties des schémas correspondant à des opérations sur l'état :

Opération

$entrée? : TYPE1$

$sortie! : TYPE2$

...

Une deuxième utilisation consiste à préciser la totalité des signifiés possibles. Par exemple nous pouvons définir l'ensemble des états possibles d'un système comme étant composé des éléments *actif* et *inactif*. Cela revient à déclarer le type libre $\acute{E}TAT$ tel que :

$$\acute{E}TAT ::= actif \mid inactif$$

Les signifiés possibles sont donc fixés une fois pour toute. Il peut être également nécessaire de les fixer temporairement, pour une opération. L'opération d'initialisation utilise constamment cette possibilité. Généralement dans l'état initial, les modèles sont vides : les ensembles représentant les signifiés sont donc des ensembles vides. Cependant rien n'exclut de déclarer aussi des ensembles en extension (par exemple $A = \{a, b, c\}$). La déclaration d'un ensemble en extension peut aussi être utilisée dans les schémas décrivant l'état, pour fixer un sous-ensemble d'un type particulier.

Une troisième utilisation consiste à définir des signifiés par rapport à d'autres signifiés. Pour les types cela revient à utiliser le produit cartésien : $C == A \times B$ (dont les individus s'écrivent *(a,b)* avec $a : A$ et $b : B$). Quant aux ensembles ils peuvent être définis à l'aide des opérateurs ensemblistes ($\cup, \cap$ ...) et logiques ($\wedge, \vee$ ...) : $C = A \cup B$, $A = B \Rightarrow C = D$. Ils peuvent aussi être définis à travers les propriétés de leurs individus. Par exemple, $C = A \cup B$ peut aussi s'écrire $C = \{x : X \mid x \in A \vee x \in B\}$ si A et B sont des ensembles de type X.

Les liens structurels non orientés d'association entre signifiés

Les liens structurels non orientés peuvent être modélisés par une relation entre ensembles. Reprenons l'exemple du langage entité/relation [Chen76] dans lequel l'arc entre une entité et une relation n'est pas orienté. Les arcs entre entités et relations sont étiquetés par les cardinalités minimales et maximales. Les cardinalités représentent le nombre de relations auxquelles peut participer une entité. La figure 3.2 présente un exemple que nous allons traiter (pour simplifier notre discours nous ne tenons pas compte des cardinalités). Sur cet exemple, quels sont les signifiés et quels sont les liens entre signifiés ?

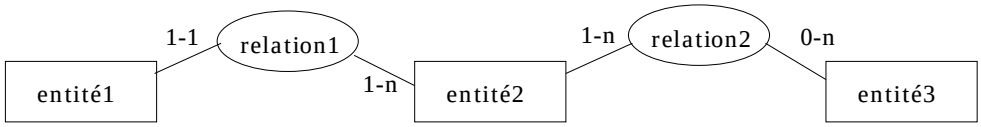


FIG. 3.2: *Exemple de modèle entité/relation à méta-modéliser*

Une première possibilité est de ne considérer qu'un type de signifiés, le type *ENTITÉ*, chaque relation étant alors le lien entre deux signifiés. Le type concerné doit être déclaré précédemment :

$[ENTITÉ]$
$lien : ENTITÉ \leftrightarrow ENTITÉ$

L'ensemble *lien* correspondant à notre figure serait :

$lien = \{(entité1, entité2), (entité2, entité3)\}$

Cette modélisation ne permet pas de nommer les relations, alors que ce nom est primordial pour la compréhension de chacune de ces relations.

Une autre possibilité consiste à considérer le type *ENTITÉ* et le type *RELATION*. L'arc reliant une entité à une relation représente alors le lien structurel entre ces signifiés :

$[ENTITÉ, RELATION]$
$lien : ENTITÉ \leftrightarrow RELATION$

Cette spécification n'est pas suffisante car elle permet de déclarer par exemple :

$lien = \{(entité1, relation1)\}$

Le langage Z permet d'exprimer de façon très précise les contraintes sur des relations. Sur

cet exemple nous pourrions spécifier que tout individu de type *RELATION* est relié à deux individus de type *ENTITÉ* par la relation *lien* :

$$\forall r : RELATION \bullet \#lien^{\sim}(\!| \{r\} |\!) = 2$$

Cependant cette modélisation n'est pas satisfaisante non plus car le signe d'un même individu de type *RELATION* peut apparaître plusieurs fois dans le modèle. Ainsi l'ensemble :

$$lien = \{(entité1, relation1), (entité2, relation1)\}$$

peut avoir de multiples représentations :

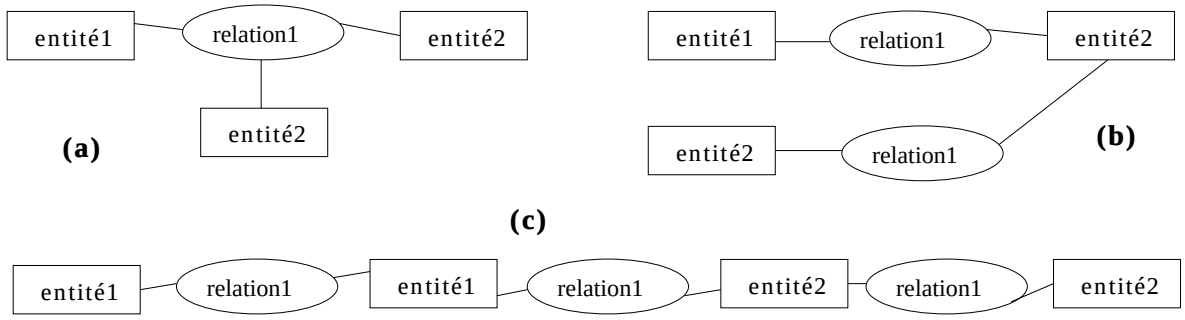


FIG. 3.3: *Représentations possibles de l'ensemble lien*

Une troisième possibilité est l'utilisation de deux types, *ENTITÉ* et *RELATION*, et d'une double relation entre individus de type *ENTITÉ* et *RELATION* :

$$[ENTITÉ, RELATION]$$
$$lien : ENTITÉ \leftrightarrow RELATION \leftrightarrow ENTITÉ$$

L'ensemble lien correspondant au modèle de la figure 3.2 est :

$$lien = \{(entité1, relation1, entité2), (entité2, relation2, entité3)\}$$

Cette modélisation impose d'elle-même à chaque individu de type *RELATION* de participer à des relations entre deux individus de type *ENTITÉ*. Elle permet également à chaque individu de type *RELATION* d'apparaître plusieurs fois dans un modèle.

Ces différentes possibilités nous ont permis de montrer que le langage Z est apte à être utilisé pour méta-modéliser tout type de lien non orienté entre signifiés. Elles nous ont également montré qu'un lien n'est pas systématiquement associé à un arc, et que la définition des signifiés et des liens nécessite un travail d'analyse rigoureux.

Les liens structurels orientés d'association entre signifiés

Les liens orientés sont aussi modélisés par des relations ou des fonctions. Le langage Z ne dispose pas de symbole différent pour distinguer un lien non orienté d'un lien orienté. La différenciation, dans un méta-modèle en Z, entre lien orienté ou non orienté se fait donc par l'interprétation du nom de la relation ou de la fonction utilisée. Pour la méta-modélisation d'un langage dans lequel il n'existe qu'une sorte de lien orienté entre deux types donnés, il n'y a aucune ambiguïté. Ce n'est plus le cas dans un langage pour lequel il existe pour deux types particuliers deux sortes de liens. Prenons l'exemple du Réseau de Petri ordinaire. Supposons que nous ayons défini les deux types $PLACE$ et $TRANSITION$. Il existe deux sortes d'arcs orientés : les arcs orientés d'une place vers une transition (que nous appellerons arbitrairement $arcPT$), et ceux orientés d'une transition vers une place ($arcTP$ par exemple). Chacun de ces arcs peut être défini comme une relation de type $PLACE \leftrightarrow TRANSITION$ ou de type $TRANSITION \leftrightarrow PLACE$. Cependant, pour bien montrer l'orientation des arcs, nous ordonnons les types des relations. Le premier type correspond à la source de l'arc, le second à sa fin : $arcTP : TRANSITION \leftrightarrow PLACE$ et $arcPT : PLACE \leftrightarrow TRANSITION$. Nous avons donc différencié les deux types d'arcs à travers le nom du lien, mais aussi à travers l'ordre de déclaration des types. Les termes utilisés pour définir un lien, ainsi que l'ordre des types utilisés dans ces liens, sont donc très importants pour comprendre leur signification, par rapport au langage modélisé. Le méta-modèle doit donc être accompagné d'un dictionnaire, qui viendra expliquer ces définitions.

$$arcTP : TRANSITION \leftrightarrow PLACE$$
$$arcPT : PLACE \leftrightarrow TRANSITION$$

Prenons un exemple de RdP ordinaire :

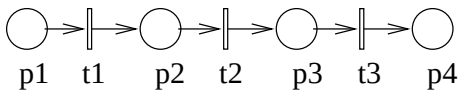


FIG. 3.4: *Exemple deRdP ordinaire*

Nous avons :

$$arcTP = \{(t1, p2), (t2, p3), (t3, p4)\}$$
$$arcPT = \{(p1, t1), (p2, t2), (p3, t3)\}$$

Les liens structurels de composition entre signifiés

Comme les deux précédents, le lien de composition est aussi modélisé par une relation ou une fonction. Un lien de composition a souvent une structure d'arbre, où un nœud représente un signifié et un arc un lien. Cette structure est alors représentée par une fonction. Par exemple, dans les actigrammes de SADT, chaque activité n'a au plus qu'une activité de niveau supérieur : l'activité de niveau père. La fonction *Père_De* permet de modéliser ce lien structurel :

$$Père\_De : ACTIVITÉ \nrightarrow ACTIVITÉ$$

Cette définition permet en outre de définir l'activité *A-0* (en supposant qu'il n'existe pas de niveau *A-1*) comme étant l'activité n'ayant pas d'activité de niveau supérieur et les activités de niveau le plus bas (l'ensemble *inf*) comme l'ensemble des activités n'appartenant pas à un niveau père d'autres activités :

$$activité = A\text{-}0 \Leftrightarrow Père\_De(activité) = \{\}$$
$$inf = \{a : ACTIVITÉ \mid a \notin \mathrm{ran}\, Père\_De\}$$

Les contraintes d'utilisation des signifiés et des liens

Comme pour la définition des signifiés, c'est la logique des prédicats qui va nous permettre d'exprimer les contraintes sur les signifiés et les liens entre signifiés.

Par exemple, chaque actigramme SADT doit contenir entre 3 et 6 activités. Comme nous l'avons exprimé dans le chapitre précédent, cette contrainte est une contrainte qui sera vérifiée en fin de construction de modèle. Notre méta-modèle d'actigrammes SADT sera donc structuré par plusieurs schémas Z. Un premier schéma définira l'état de notre spécification, plusieurs schémas définiront ensuite les opérations de construction d'un modèle, viendra alors un schéma de vérification du modèle qui permettra d'exprimer les contraintes sur les signifiés et les liens.

Cette contrainte du nombre d'activités par niveau n'est pas vraie pour tous les actigrammes : ce n'est pas le cas du niveau *A-0*. Pour exprimer cette contrainte, nous réutilisons la fonction définie précédemment (*Père_De*) : *Père_De(a)* représente l'activité père de a, et $(Père\_De^{\sim}(Père\_De(a)))$ l'ensemble des activités de même niveau que a (a y compris). $\#(Père\_De^{\sim}(Père\_De(a)))$ représente le nombre d'activité du même niveau que a. La condition à respecter s'exprime sous la forme :

$$\forall\, a : ACTIVITÉ \mid a \neq A\text{-}0 \Rightarrow 3 \leq \#(Père\_De^{\sim}(Père\_De(a))) \leq 6$$

Nous voyons à travers cet exemple que le langage Z est bien adapté à l'expression de contraintes sur les signifiés et les liens. Grâce aux opérateurs de la théorie des ensembles ou de la logique du premier ordre, le langage Z permet d'exprimer beaucoup plus de contraintes que le langage entité/relation couramment employé. D'autres langages, OMDGA [Couffin97] par exemple, permettent aussi d'exprimer ces contraintes. L'avantage de Z est de pouvoir les exprimer avec les mêmes opérateurs, et au sein des mêmes structures - les schémas - que les définitions des signifiés et des liens [Pietrac et al.96].

3.3.2 La syntaxe concrète

Les signes

Nous avons déjà dit que l'étude de la syntaxe concrète ne faisait pas partie de notre étude. Cependant certaines pistes peuvent tout de même être envisagées. Le langage Z ne permet pas de représenter les signes. Cependant la forme des signes, leur style ... pourront être exprimés à travers leur nom : cercle, carré ... La description textuelle d'un signe augmente énormément la taille du méta-modèle. La description graphique d'un arc par exemple peut nécessiter la description du type de ligne employé, sa couleur, les terminaisons employées à chaque extrémité, les coordonnées de chaque extrémité, etc. Le schéma Z suivant correspond à la description générique des arcs selon [Gee95] :

GenericArcs

$$
\begin{array}{l}
arcs : \mathbb{P}\, Arc \\
name : Arc \twoheadrightarrow Names \\
shape : Arc \twoheadrightarrow Arc\text{-}Shapes \\
line : Arc \twoheadrightarrow Lines \\
type : Arc \twoheadrightarrow Arc\text{-}types \\
from\text{-}n : Arc \twoheadrightarrow Nodes \\
to\text{-}n : Arc \twoheadrightarrow Nodes \\
from\text{-}end : Arc \twoheadrightarrow Ends \\
to\text{-}end : Arc \twoheadrightarrow Ends \\
start : Arc \twoheadrightarrow Point \\
finish : Arc \twoheadrightarrow Point \\
path : Arc \twoheadrightarrow \mathbb{P}\, Point \\
direction : Arc \twoheadrightarrow Boolean \\
\hline
\forall\, a : Arc \mid a \in arcs \bullet start(a) \in path(a) \land finish(a) \in path(a) \\
\forall\, a : Arc \mid a \in arcs \bullet start(a) \in path(from\text{-}n) \land finish(a) \in path(to\text{-}n)
\end{array}
$$

Les liens entre signes et signifiés

Prenons l'exemple du langage entité/relation (figure 3.5). Un premier méta-modèle de ce langage, qui aura un point de vue syntaxe abstraite et sémantique, représente les signifiés par des ensembles et leurs éléments regroupés au sein d'un schéma qui représente

l'état. Pour prendre en compte tous les attributs de la syntaxe concrète, un nouveau méta-modèle doit être construit. Il peut être déduit du premier en modélisant chaque signifié du premier méta-modèle par un schéma dans le deuxième. Chacun de ces schémas (qui représente chacun un signifié) contient la description des signes sous forme d'ensembles et d'éléments d'ensembles. Chacun de ces schémas sera alors utilisé comme un type, les opérations seront transformées pour permettre d'instancier ces schémas pour construire un modèle.

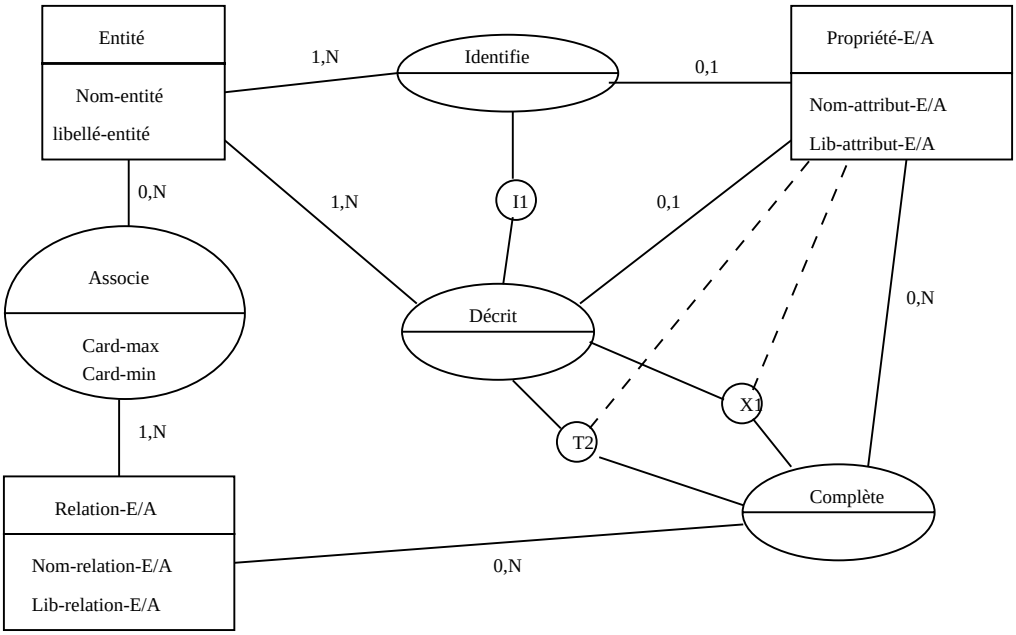


FIG. 3.5: *Méta-modèle du langage Entité/relation, d'après [Kiefer96]*

Le premier méta-modèle aura comme schéma représentant l'état d'un modèle exprimé en langage Entité/relation (en supposant que les types ont été définis) :

Entité/Relation

...

entité : *ENTITÉ*

...

Tandis que le deuxième méta-modèle contiendra les schémas :

Entité

identifiant : *NOM*
forme : *FORME*
x, y : *POSITION*
u, v : *TAILLE*

---

forme = *rectangle*

Entité/Relation

...
entité : *Entité*
...

L'atelier MCD de Graphtalk propose une extension du langage Entité/relation. Pour éviter d'obtenir des modèles dans lesquels les liens se coupent, une entité peut être représentée deux fois sur le même modèle. Ces deux représentations de la même entité sont marquées d'un point pour que le lecteur remarque cette duplication plus facilement. Dans ce cas, un seul signifié est associé à deux signes. Notre méta-modèle est tout à fait utilisable dans ce cas (même identifiant, positions différentes) sous la condition bien sûr de prévoir plusieurs formes possibles (*rectangle* et *rectangle+point* par exemple).

3.3.3 La sémantique interne

Liens sémantiques entre signifiés du langage

Le chapitre 2 (« L'étude de la sémantique interne », section 2.2.2 page 47) a montré que l'étude des liens sémantiques entre signifiés est très proche de l'étude de la syntaxe abstraite, la distinction entre ces deux types de liens ne reposant que sur la désignation des liens. La description des liens sémantiques entre signifiés du langage se fait donc de la même façon que la description des liens syntaxiques entre signifiés du langage : par l'utilisation de relations et de fonctions.

Prise en compte du temps

Nous avons déjà dit, dans le chapitre précédent, que le temps peut être pris en compte explicitement ou transparaître sous forme de séquentialité dans la sémantique interne du langage. Plus précisément, les variables utilisées pour caractériser le temps peuvent être

de trois formes :

- continues : elles évoluent sur un intervalle de nombres réels ;
- discrètes : elles évoluent sur un intervalle de nombres entiers ;
- symboliques : elles décrivent une chronologie, cependant elles ne sont pas utilisées pour définir des dates.

Le langage Z ne contient pas le type $\mathbb{R}$, cependant plusieurs auteurs proposent d'étendre la boîte à outils mathématique de Z par l'inclusion de ce type [Barden et al.94], [Valentine95]. Outre l'ensemble $\mathbb{R}$, ces auteurs introduisent de nombreux opérateurs de calcul sur $\mathbb{R}$: addition, multiplication, intervalles, puissance, etc. En incluant $\mathbb{R}$, les variables continues peuvent être modélisées comme des fonctions de $\mathbb{R}$ vers un autre ensemble. Prenons l'exemple de variables booléennes. Ces variables prennent leur valeur sur l'ensemble $\mathbb{B}$, qui n'est pas un type en Z mais qui peut être défini de la manière suivante : $\mathbb{B} = \{n : \mathbb{N} \mid n = 0 \lor n = 1\}$. Si l'on considère que ces variables sont continues, elles peuvent être définies comme des éléments du type : $VAR\_BOOL == \mathbb{R} \rightarrow \mathbb{B}$.

Le type $\mathbb{N}$ fait partie des types du langage Z. Considérons maintenant que les variables booléennes de l'exemple précédent sont des variables entières. Elles peuvent être modélisées comme étant des éléments du type : $VAR\_BOOL == \mathbb{N} \rightarrow \mathbb{B}$. Une autre possibilité est d'utiliser les suites, ce qui revient à écrire $VAR\_BOOL == \mathrm{seq}\,\mathbb{B}$. Ces deux possibilités sont équivalentes bien qu'à notre avis la première soit plus indépendante de l'implantation que la seconde (la transformation ensemble $\rightarrow$ séquence $\rightarrow$ liste ou tableau permet de passer de la spécification au code ...).

Ces deux possibilités permettent de définir une évolution dans le temps, des dates, des durées. L'évolution du temps peut être spécifiée par le fonctionnement d'une horloge qui incrémente le temps, qu'il soit continu [Stoddart et al.95] ou discret [Bowen96] (chapitre 9).

Le langage Z permet de spécifier les changements d'états par l'utilisation d'une convention de décoration des variables : l'apostrophe. Cette convention peut être utilisée pour les variables symboliques. Si nous reprenons notre exemple précédent, nous pouvons écrire $VAR\_BOOL == \mathbb{B}$. Si nous avons une variable a de type VAR_BOOL, la chronologie d'évolution de la variable a sera représentée par a (l'état avant) et a' (l'état après).

Le langage Z, à condition d'y inclure l'ensemble $\mathbb{R}$ et ses opérateurs, permet de clairement distinguer les trois types de variables temporelles et leur évolution.

3.3.4 La sémantique externe

Concepts de la théorie et liens sémantiques entre langage et théorie

Comme les descriptions de la syntaxe ou de la sémantique interne, la description de la sémantique externe est basée sur l'utilisation des ensembles et relations. Ainsi :

- la modélisation des signifiés de la théorie est identique à la modélisation des signifiés du langage ;

- la modélisation des signifiants de la théorie et des liens sémantiques entre signifiés du langage et de la théorie est identique à la modélisation des liens - syntaxiques (non ordonnés, ordonnés ou structurels) ou sémantiques - entre signifiés du langage.

Il est cependant souhaitable de distinguer dans le méta-modèle la partie décrivant l'instanciation du langage de celle décrivant l'instanciation de la théorie. Cela peut être fait en scindant l'état du système décrit en deux schémas : un schéma décrivant le langage, et un schéma incluant le premier et y ajoutant la description des concepts de la théorie ainsi que les liens entre signifiés du langage et de la théorie.

Prenons l'exemple du grafcet. Un concept indispensable à la modélisation d'un SAP avec le langage grafcet est la *situation*. Si nous prenons *situation* au sens de [Blanchard79] et [Afnor93] une situation correspond à l'ensemble des étapes actives d'un modèle grafcet. Le signifié *situation* n'a pas de signe associé, ce qui est normal car il ne correspond pas à un individu mais à un ensemble d'individus faisant partie du langage : *situation* fait donc partie du langage grafcet. A cette situation, nous pouvons faire correspondre l'état (*état_système*) du système (*système*) modélisé. Les signifiés *état_système* et *système* ne correspondent pas directement à des signifiés du langage, ils n'ont pas de signes associés : ce sont donc des concepts de la théorie associée au langage grafcet (utilisé pour la modélisation des SAP).

Considérons l'extrait du schéma *État_langage_grafcet* décrivant l'état d'un modèle grafcet :

$$\acute{E}TAT ::= active \mid inactive$$

État_langage_grafcet

$\ldots$

$étape : \mathbb{F}\ ÉTAPE$

$état : ÉTAPE \pfun ÉTAT$

$situation : \mathbb{F}\ ÉTAPE$

$\ldots$

$\ldots$

$\forall\, e : ÉTAPE \bullet état(e) = active \Leftrightarrow e \in situation$

$\ldots$

Le schéma *État_théorie_grafcet* décrivant la théorie associée serait :

État_théorie_grafcet

État_langage_grafcet

$système : SYSTÈME$

$état\_système : SYSTÈME \pfun \mathbb{F}\ ÉTAPE$

$état\_système(système) = situation$

Les instances possibles et impossibles des signifiés et des liens entre signifiés

Pour définir l'ensemble des instances possibles d'un signifié, la première possibilité est de définir un type libre. Définir un type libre consiste à déclarer un type et à donner toutes ses instances possibles. Cette solution implique que toutes les instances de tous les modèles appartiendront à cet exemple. Ceci est donc vrai quel que soit le système modélisé. Le type libre sera donc adapté lorsque le langage est prévu pour la modélisation d'une classe bien définie de systèmes.

Une deuxième possibilité est d'énumérer dans un schéma l'ensemble des instances possibles d'un ensemble. Dans le schéma spécifiant l'état du langage, ceci sera adapté à tous les modèles construits. Dans le schéma spécifiant l'état de la théorie, cela permettra de restreindre l'utilisation du langage et donc de l'adapter aux systèmes modélisés. De la même façon, il est possible d'énumérer des individus qui ne doivent pas être utilisés, d'écrire des prédicats restreignant l'utilisation de relations ou de fonctions ... bref d'adapter le langage au système.

3.4 Méta-modélisation des méthodes

3.4.1 Les opérations

Les opérations se modélisent par l'intermédiaire d'un schéma. Les opérations peuvent se classer en deux catégories suivant qu'elles modifient ou non l'état de la spécification. Les opérations de construction, d'importation, d'exportation, d'interprétation et de simulation modifient l'état. Seules les opérations de vérification et de validation ne le modifient pas. Nous avons déjà dit que les schémas décrivant une opération doivent inclure la déclaration de l'état du système avant et après l'opération (utilisation de $\Xi \acute{E}tat$ ou de $\Delta \acute{E}tat$, voir section 3.2.3 page 66). Les opérations peuvent, en outre, nécessiter une ou plusieurs entrées fournies par le concepteur, ou générer une ou plusieurs sorties qui lui sont destinées. Les entrées et sorties d'un schéma ne font pas partie de l'état du système décrit. La désignation des entrées est décorée par un point d'interrogation (entrée?), la désignation des sorties est décorée d'un point d'exclamation (sortie!). Par exemple :

État

Partie déclarative

Partie prédicative

Opération1

$\Delta \acute{E}tat$
Entrée?
Sortie!

Prédicat spécifiant la condition sur l'entrée $\wedge$
Prédicats spécifiant l'état des variables après opération $\wedge$
Prédicat spécifiant l'état de la sortie

Les opérations de construction

Une opération de construction est modélisée en Z par un schéma ayant au moins une entrée qui viendra s'ajouter à l'état du système spécifié. La partie déclarative du schéma minimum comprend donc cette entrée et les deux états différents du système (soit $\Delta Etat$). La partie prédicative comprend au moins la déclaration des ensembles de l'état après modification.

Cette opération correspond à l'utilisation « courante » du langage Z : l'ajout d'une

donnée dans une base. L'exemple suivant, extrait de [Diller94], d'ajout d'un numéro de téléphone (schéma *PhoneDB*) dans une liste correspond à cette utilisation :

PhoneDB

$$members : \mathbb{P}\, Person$$
$$telephones : Person \leftrightarrow Phone$$
$$\hline$$
$$\mathrm{dom}\, telephones \subseteq members$$

AddEntry

$$\Delta PhoneDB$$
$$name? : Person$$
$$newnumber? : Phone$$
$$\hline$$
$$name? \in members$$
$$name? \mapsto newnumber? \notin telephones$$
$$telephones' = telephones \cup \{name? \mapsto newnumber?\}$$
$$members' = members$$

Les opérations de vérification

Les opérations de vérification consistent à s'assurer que certaines contraintes d'utilisation des signifiés et des liens sont respectées dans un modèle. Nous avons déjà présenté, dans la section 3.3.1 « Les contraintes d'utilisation des signifés et des liens », l'utilisation du langage Z pour modéliser ces contraintes. Ces contraintes doivent être exprimées au sein d'un schéma qui correspond à une opération ne modifiant pas l'état du système. Dans le cas déjà présenté (section 3.3.1) des actigrammes SADT devant comporter entre trois et six activités, cela pourrait se présenter sous la forme :

$$RÉPONSE ::= manque \mid correct \mid trop$$

Vérif

$$\Xi SADT$$
$$a? : ACTIVITÉ$$
$$rép! : RÉPONSE$$
$$\hline$$
$$\#Père\_De^{\sim}(a?) < 3 \land rép! = manque$$
$$\lor$$
$$\#Père\_De^{\sim}(a?) > 6 \land rép! = trop$$
$$\lor$$
$$6 \geq \#Père\_De^{\sim}(a?) \geq 3 \land rép! = correct$$

Le schéma *Vérif* permet de vérifier qu'un actigramme donné $a?$ comprend un nombre correct d'activité, c'est-à-dire entre 3 et 6. S'il en comprend moins de 3, la réponse *manque* est donnée pour signifier qu'il doit être complété. Si plus de 6 activités sont présentes, le message *trop* indique qu'il faut en supprimer. Ce schéma décrit une opération ne modifiant

pas l'état du schéma *SADT* qui décrit l'état du système.

Les opérations d'importation et d'exportation

Comme nous l'avons déjà dit dans le chapitre 3, François Kiefer a réalisé dans [Kiefer96] un travail exhaustif sur les différents cas possibles de correspondance sémantique entre signifiés de deux langages. Du point de vue de notre méta-modèle, la distinction entre l'intégration de deux langages par exportation ou par importation n'a pas lieu d'être. En effet, ces deux formes d'intégration seront spécifiées par une opération d'ajout d'un élément dans un modèle. L'exemple suivant spécifie l'opération d'exportation (ou d'importation) dans le cas de correspondance directe entre concepts. Cette opération consiste à ajouter un élément dans un modèle exprimé dans un langage Y, en spécifiant en même temps que cet élément a une correspondance sémantique avec un élément du modèle exprimé dans le langage X.

$[ConceptX\,1, ConceptY\,1]$

LangageX

$$\begin{array}{l} \ldots \\ ensembleX\,1 : \mathbb{P}\, ConceptX\,1 \\ \ldots \end{array}$$

LangageY

$$\begin{array}{l} \ldots \\ ensembleY\,1 : \mathbb{P}\, ConceptY\,1 \\ \ldots \end{array}$$

Integration

$$\begin{array}{l} LangageX \\ LangageY \\ correspondance : ConceptX\,1 \rightarrow ConceptY\,1 \end{array}$$

ExportationDeXversY

$$\begin{array}{l} \Delta Integration \\ in? : ConceptX\,1 \\ new? : ConceptY\,1 \\ \hline in? \in ensembleX\,1 \\ in? \notin \mathrm{dom}\, correspondance \\ new? \notin ensembleY\,1 \\ ensembleX\,1' = ensembleX\,1 \\ ensembleY\,1' = ensembleY\,1 \cup \{new?\} \\ correspondance'(in?) = new? \end{array}$$

$ImportationDeYdepuisX \mathrel{\widehat{=}} ExportationDeXversY$

Dans cet exemple simple, nous avons spécifié que la création d'un nouvel élément, dans le modèle exprimé avec le langage Y, ne peut avoir lieu que si :

- un élément de correspondance directe existe dans le modèle exprimé avec le langage X ;
- cet élément du modèle exprimé avec le langage X n'a pas déjà un élément correspondant dans le modèle exprimé avec le langage Y ;
- cet élément n'existe pas déjà.

Outre les conditions à vérifier en cours de construction du modèle, il est nécessaire de vérifier d'autres conditions en fin de construction. Ainsi lorsque le système modélisé dans le langage X est le même ou est un sous-système de celui modélisé dans le langage Y (et dans le cas de la correspondance directe), il faut vérifier qu'à tout élément de $ensembleX1$ correspond un et un seul élément de $ensembleY1$. Nous dirons alors que le modèle est complet. Ceci peut être spécifié par le schéma suivant :

$REPONSE ::= complet \mid incomplet$

Verif

$$\Xi Integration$$
$$rep! : REPONSE$$

$$\begin{aligned}&((\mathrm{dom}\, correspondance = ensembleX1 \land \mathrm{ran}\, correspondance = ensembleY1)\\ &\qquad \land rep! = complet)\\ &\qquad\quad \lor\\ &((\mathrm{dom}\, correspondance \neq ensembleX1 \lor \mathrm{ran}\, correspondance \neq ensembleY1)\\ &\qquad \land rep! = incomplet)\end{aligned}$$

Cette correspondance directe est le cas le plus simple de correspondance. Dans les cas de correspondance indirecte, la démarche est la même. Il s'agit alors de spécifier plusieurs relations de correspondance entre éléments des deux modèles.

Les opérations d'interprétation

Les opérations d'interprétation consistent à exprimer des propriétés dans le langage qui permettra de les valider. La plupart du temps, ces propriétés sont d'abord exprimées dans le langage courant, le français dans notre cas. L'opération d'interprétation ne pourrait donc être méta-modélisée que si le français et le langage cible avaient auparavant été méta-modélisés. L'étude de la structure du français ne rentrant pas dans le cadre de notre travail, l'opération d'interprétation ne sera pas méta-modélisée.

Par contre, il est important d'exprimer les propriétés des modèles afin de pouvoir les valider. Dans notre approche de méta-modélisation, cela consiste à exprimer ces propriétés dans le langage de méta-modélisation, en utilisant les ensembles et individus décrivant le modèle à valider. La théorie des ensembles et la logique des prédicats, utilisées dans le langage Z, permettent d'exprimer les propriétés à valider.

Les opérations de validation

Les opérations de validation peuvent être décrites dans deux contextes différents :

- la propriété à valider est particulière à un modèle donné, cette propriété n'est pas décrite dans le méta-modèle ;
- la propriété à valider est plus générale, elle est validable pour un ensemble de modèles : elle est décrite dans le méta-modèle.

Dans le premier cas, il s'agit donc de valider une propriété d'une instance du méta-modèle. La démonstration de cette propriété s'écrit sous la forme de la démonstration d'une proposition, identique à n'importe qu'elle démonstration de proposition écrite en langage Z.

Dans le deuxième cas, la propriété à valider est spécifiée au sein d'un schéma décrivant une opération ne modifiant pas l'état, sous la forme d'un prédicat. Spécifier de cette manière une propriété ne pose pas de problème, même si la propriété est invérifiable ... Si l'on désire aller plus loin et spécifier de quelle manière est validée la propriété, il faut d'abord spécifer une opération d'exportation vers un modèle d'un autre langage, modèle sur lequel une propriété équivalente pourra être vérifiée. Ceci est le cas, par exemple, de la démarche utilisée dans AGGLAE (chapitre 2). Cela est aussi le cas pour le RdP lorsqu'il est traduit en matrice pour valider ses propriétés.

Les opérations de jeu

La démarche de base utilisée pour modéliser une opération de jeu est très proche de celle utilisée dans les opérations de construction, d'importation ou d'exportation. Elle est basée sur une modification de l'état du système décrit. Pour les réseaux de Petri (chapitre 4), l'opération de tir d'une transition est spécifiée par un schéma comprenant une entrée – la transition qui doit être tirée, avec une condition à respecter – et les ensembles et individus après opération. Pour jouer le RdP, il faut donc réaliser plusieurs fois l'opération de tir.

Pour certains langages, l'opération de jeu consiste à réaliser une sous-opération jusqu'à obtenir une propriété donnée : le jeu avec recherche de stabilité du Grafcet par exemple.

Dans ce cas, cette opération est spécifiée grâce à un schéma contenant l'opérateur existentiel. L'exemple suivant, extrait de [Spivey94], décrit la recherche (schéma *FindBirthday*1) d'une date d'anniversaire dans un tableau (schéma *BirthdayBook*1).

*BirthdayBook*1

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}$$

$$\forall i, j : 1..hwm \bullet i \neq j \Rightarrow names(i) \neq names(j)$$

*FindBirthday*1

$$\Xi BirthdayBook1$$
$$name? : NAME$$
$$date! : DATE$$

$$\exists i : 1..hwm \bullet name? = names(i) \land date! = dates(i)$$

L'opérateur existentiel permet ici de spécifier la recherche, sans avoir à spécifier une succession de sous-opérations de lecture dans le tableau. Cette opération correspond pourtant à une implantation de sous-opérations automatiques, qui dans un langage tel que le Pascal serait par exemple codée de la façon suivante :

```
procedure FindBirthday(name : NAME; var date : DATE);
        var i : INTEGER;
begin
        i := 1;
        while names[i] ≠ name do i := i+1;
        date := dates[i]
end;
```

Avec le langage Z, la spécification des opérations de jeu ne nécessite donc pas la spécification de boucles, retours et branchements divers. Seule la condition finale à respecter est spécifiée.

3.4.2 La démarche d'intégration des opérations

L'utilisation courante du langage Z consiste à spécifier des opérations, indépendantes les unes des autres, sur un état. Ce langage n'a pas été conçu pour modéliser une suite d'opérations ordonnées dans le temps. Cependant, deux opérateurs existent pour définir la séquentialité de deux opérations : la composition séquentielle ($A \fatsemi B$) et le tubage ($A \gg B$).

Soit le schéma C défini comme la composition séquentielle des schémas A et B : $C \mathrel{\widehat=} A \fatsemi B$ (les schémas A et B décrivent des opérations sur le schéma État). Dans le schéma C, l'état des variables de ÉTAT avant opération (resp. après opération) est égal à celui

avant (resp. après) l'opération A (resp. B). L'état des variables après opération A est égal à celui avant opération B. Les entrées (resp. les sorties) de C sont les entrées (resp. les sorties) de A et de B. Sur un exemple simple :

État

$x, y : \mathbb{N}$

A

$\Delta État$

$a1?, a2! : \mathbb{N}$

$x' = 2 \land y' = y + a1? \land a2! = y'$

B

$\Delta État$

$b1?, b2! : \mathbb{N}$

$x' = x + b1? \land y' = y + 3 \land b2! = y'$

Le schéma $C \mathrel{\widehat{=}} A \fatsemi B$ est donc :

C

$\Delta État$

$a1?, a2!, b1?, b2! : \mathbb{N}$

$x' = 2 + b1? \land y' = y + a1? + 3 \land a2! = y + a1? \land b2! = y + a1?$

Soit le schéma F égal au tubage des schémas D et E : $F \mathrel{\widehat{=}} D \gg E$. Les entrées (resp. les sorties) de F sont les entrées (resp. les sorties) de D (resp. de E). Par contre, les sorties de D sont égales aux entrées de E. Les états des variables de État sont les mêmes dans F que dans D et E. Sur un autre exemple :

D

$x, x' : \mathbb{N}$

$a?, b! : \mathbb{N}$

$x' = x + a? \land b! = x' + 2$

E

$y, y' : \mathbb{N}$

$b?, c! : \mathbb{N}$

$y' = y + 3 \land c! = y' + b?$

Le schéma $F \mathrel{\widehat{=}} D \gg E$ est donc :

$$
\begin{array}{|l}
\hline
F \\
x, x', y, y' : \mathbb{N} \\
a?, c! : \mathbb{N} \\
\hline
x' = x + a? \land y' = y + 3 \land c! = y' + x' + 2 \\
\hline
\end{array}
$$

Cet exemple nous permet de montrer que l'opérateur de tubage n'est pas utilisable lorsque les deux schémas tubés spécifient une opération sur un état. En effet les deux schémas comportent des opérations différentes sur les mêmes variables et ne peuvent donc pas être tubés. L'opérateur de composition séquentielle peut tout à fait être utilisé, à condition que le choix d'une des entrées de la deuxième opération composée ne dépende pas du résultat de la première opération.

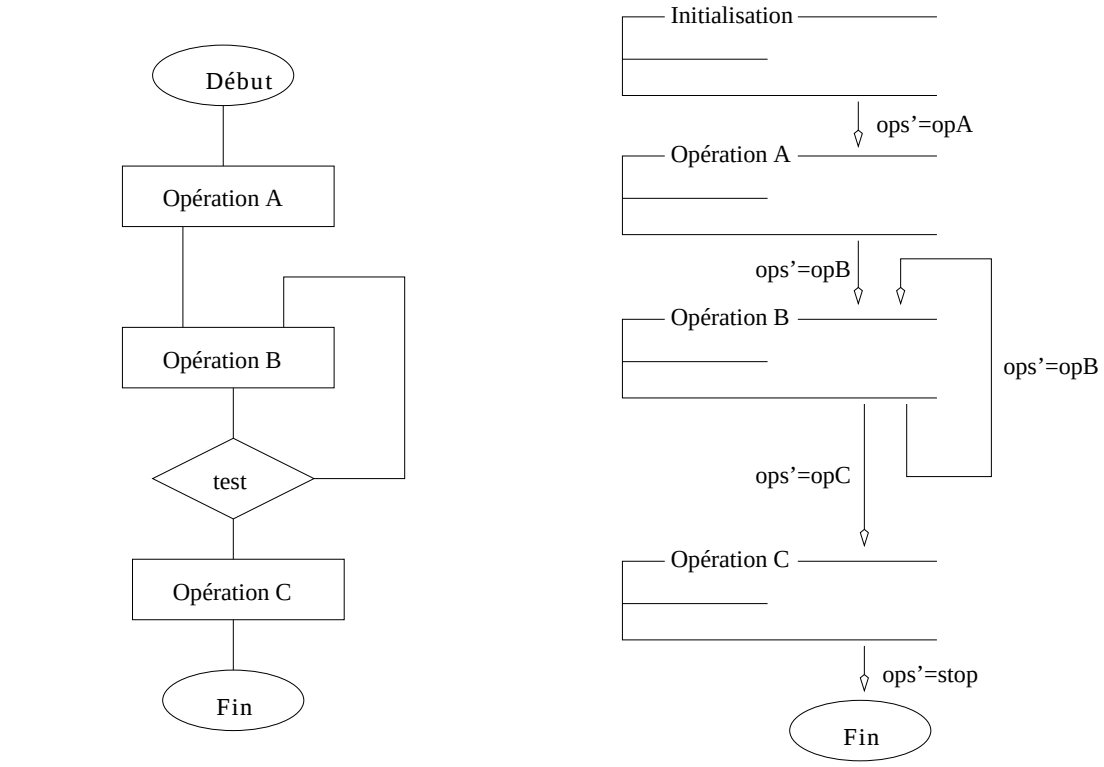
Dans la plupart des méthodes, la démarche de construction d'un modèle n'est pas figée. Si toutes les opérations de construction sont toujours autorisées, la démarche d'intégration est inexistante : les opérations sont modélisées par des schémas, il n'y a pas de condition d'utilisation des schémas. Si certaines opérations ne sont possibles qu'à certaines conditions, ces conditions doivent être exprimées à l'aide d'un prédicat écrit dans le schéma décrivant l'opération concernée. Ces conditions à vérifier permettent notamment d'exprimer deux types de séquentialité entre opération : une séquentialité algorithmique et une séquentialité pratique que nous allons définir maintenant.

La séquentialité algorithmique

Lorsque la démarche d'intégration impose un ordonnancement figé des opérations, avec éventuellement des choix possibles, nous parlons de séquentialité algorithmique. Cette séquentialité peut, par exemple, être représentée sous la forme d'un organigramme de traitement (exemple figure 3.6 (a) [Afnor75]).

Pour modéliser cette séquentialité dans un modèle Z, nous utilisons une variable spécifique qui va changer de valeur dans les schémas décrivant les opérations, autorisant ou non l'exécution d'autres opérations. La figure 3.6 (b) représente cette séquentialité de schéma Z. La variable *ops* définit l'opération (le schéma) qui pourra être exécutée. Elle peut prendre différentes valeurs : opA, opB, opC ou stop. Le schéma « OpérationA » par exemple doit avoir dans sa précondition $ops = opA$, et contenir dans sa postcondition $ops' = opB$. L'opération B est un cas courant d'opération pouvant avoir plusieurs résultats, avec ensuite un test sur ce résultat pour déterminer l'opération suivante à exécuter. Dans notre modèle Z, il n'y a pas un schéma ne spécifiant que ce test. En fait, comme nous venons de le voir, le test est intégré à chaque schéma : c'est sa précondition.

Notre opération B est un cas particulier dans la mesure où un cas possible du résultat du test est une nouvelle exécution de l'opération B. Il convient de remarquer que ce



(a) Organigramme de traitement [Afnor75] (b) Succession de schémas

FIG. 3.6: *exemple de séquentialité algorithmique*

cas correspond à une exécution lancée par l'utilisateur. Dans le cas d'une « exécution automatique » la succession peut être décrite à l'intérieur du schéma. Ce cas est identique à la description d'une opération de jeu construite à partir de sous-opérations répétitives.

La séquentialité pratique

Lorsque la démarche d'intégration n'impose pas d'ordonnancement figé des opérations, mais que certaines opérations ne sont pas possibles avant certaines autres, nous parlons de séquentialité pratique. Ceci est le cas, par exemple, pour les opérations de construction de graphes. L'opération de construction de liens entre deux nœuds ne peut se faire que si ces nœuds existent. Cette condition sera intégrée à la précondition du schéma décrivant cette opération de construction des liens. Ce sont donc toutes les préconditions des schémas qui vont imposer une certaine séquentialité en empêchant ou autorisant certaines opérations.

Cette approche est intéressante car elle correspond à la plupart des méthodes de construction de modèles. Elle permet de n'autoriser certaines opérations qu'à certains moments sans pour autant interdire des retours en arrière : les méthodes ont une structure générale linéaire qui cache en fait de multiples contrôles de branchement. La spécification

de schémas avec des préconditions permettant ces opérations doit être complétée par la spécification de schémas prenant en compte les cas dans lesquels ces préconditions ne sont pas satisfaites. Cette démarche progressive de spécification facilite l'analyse d'une spécification totale.

3.5 Conclusion

Dans le chapitre 2, nous avons montré que les attentes de la méta-modélisation des langages et des méthodes sont très nombreuses. Dans ce chapitre 3, nous avons présenté notre solution pour répondre à l'ensemble des besoins : l'utilisation du langage formel Z. Cette utilisation permet la méta-modélisation de la syntaxe et de la sémantique des langages, ainsi que la méta-modélisation des méthodes, y compris celle de construction et celle de jeu des modèles. Aucune autre approche de méta-modélisation n'a un aussi large champ d'action. De plus l'utilisation de Z comme méta-méta-langage permet de vérifier et de valider les méta-modèles.

De courts exemples nous ont permis de présenter des parties de ce qui pourrait être un méta-modèle en langage Z. Afin de valider complètement notre approche par rapport aux besoins de méta-modélisation, nous allons maintenant l'utiliser sur deux exemples : un langage dans le chapitre 4 et une méthode dans le chapitre 5. Le langage choisi est un langage nous permettant d'utiliser au maximum les capacités de notre approche. Il nécessite la méta-modélisation de sa syntaxe, de sa sémantique mais aussi des opérations de jeu et de construction de modèles : c'est le réseau de Petri généralisé. La méthode quant à elle est mono-modèle mais multi-langages. Nous avons, en effet, souhaité méta-modéliser une méthode de jeu d'un modèle construit à partir de plusieurs langages. Les approches existantes de méta-modélisation ne permettent pas l'étude de ce genre de méthode, ce qui prouvera tout l'intérêt de notre approche. La méthode choisie combine des réseaux de Petri et des équations différentielles et est développée au LAAS. Cette méthode nous permettra, en outre, de montrer de quelle manière notre approche peut être étendue aux systèmes hybrides.

Chapitre 4

Étude d'un langage : le RdP généralisé

Ce chapitre a pour objectif de valider notre approche par la méta-modélisation d'un langage. Le langage choisi est une classe particulière de réseau de Petri : le RdP généralisé. Ce langage permet de construire des modèles ayant un comportement dynamique : la méta-modélisation de ce langage nous permettra donc de spécifier aussi bien la syntaxe que la sémantique du langage, le joueur et la construction des modèles, la validation et la vérification du méta-modèle. A part la syntaxe concrète du langage, nous aurons ainsi rigoureusement défini chacun des aspects du langage et de son utilisation, ce qu'aucune autre approche de méta-modélisation n'est capable de faire. De plus, les RdP généralisés sont très répandus dans le domaine de la conception des SAP, ce qui nous permettra de montrer que même la méta-modélisation d'un langage très connu et considéré comme « formel » apporte une meilleure connaissance et une plus grande rigueur à la définition de ce langage. Le document de référence que nous avons choisi est un document très souvent cité en référence dans les publications sur les réseaux de Petri : l'article de synthèse de Tadao Murata [Murata89].

La première section de ce chapitre présente la définition du réseau de Petri donnée par Murata. Dans la deuxième section, chacun des points de cette définition est repris et commenté afin de présenter l'extrait de méta-modèle correspondant. Le méta-modèle du RdP généralisé est ainsi construit au fur et à mesure de la présentation de la définition et ne doit pas être considéré à ce moment là comme définitif. La section trois complète ce méta-modèle par l'étude des opérations de construction d'un modèle permettant de présenter dans la section quatre un méta-modèle qui semble complet. A travers la vérification de ce méta-modèle, la section cinq nous montrera justement les manques de ce méta-modèle qui sera ensuite validé dans la section six.

4.1 Définition de référence

La « définition formelle » proposée dans l'article de Murata est la suivante :

Un réseau de Petri est un 5-uplet, PN = (P, T, F, W, M_o) où :

$P = \{p_1, p_2, \ldots, p_n\}$ *est un ensemble fini de places,*

$T = \{t_1, t_2, \ldots, t_n\}$ *est un ensemble fini de transitions,*

$F \subseteq (P \times T) \cup (T \times P)$ *est un ensemble d'arcs*

$W : F \to \{1, 2, 3, \ldots\}$ *est la fonction poids,*

$M_0 : P \to \{1, 2, 3, \ldots\}$ *est le marquage initial,*

$P \cap T = \emptyset$ *et* $P \cup T \neq \emptyset$.

Une structure de réseau de Petri N = (P, T, F, W) sans marquage initial spécifique est noté N.

Un réseau de Petri avec un marquage initial est noté (N, M_0).

Cette définition formelle n'aborde pas l'aspect évolution du marquage d'un RdP généralisé. Cependant cet aspect est présenté dans le texte du chapitre II « transition, activation et tir » de l'article de Murata (dans lequel figure également la définition formelle). Seules sont reproduites ici les parties concernant le marquage et l'évolution des marquages d'un RdP généralisé.

Un marquage (état) assigne à chaque place un entier non négatif. Si un marquage assigne à chaque place p un entier non négatif k, nous disons que p est marqué avec k jetons. Un marquage est noté M, un m-vecteur, où m est le nombre total de places. La $p^{ième}$ *composante de M, notée M(p), est le nombre de jetons de la place p.*

En modélisation, utilisant les concepts de conditions et d'événements, les places représentent les conditions, et les transitions représentent les événements. Une transition (un événement) a un certain nombre de places d'entrée et de sortie représentant respectivement les pré-conditions et les post-conditions de l'événement.

Le comportement de certains systèmes peut être décrit en termes d'états de systèmes et de leurs changements.

Afin de simuler le comportement dynamique d'un système, un état ou marquage d'un réseau de Petri est changé suivant les règles de transition (tir) suivantes :

1° *une transition t est dite validée si chaque place p d'entrée de t est marquée avec au moins w(p,t) jetons, où w(p,t) est le poids de l'arc de p vers t ;*

2° *une transition validée doit ou ne doit pas être tirée (en fonction de l'événement qui arrive) ;*

3° *le tir d'une transition validée t retire w(p,t) jetons de chaque place d'entrée p de t, et rajoute w(t,p) jetons à chaque place de sortie p de t, où w(t,p) est le poids de chaque arc de t vers p.*

Une transition sans aucune place d'entrée est appelée une transition source, et une transition sans place de sortie est appelée transition puits. Une transition source est inconditionnellement validée, et le tir d'une transition puits consomme des jetons, mais n'en produit aucun.

4.2 Construction du méta-modèle

Les types de base sont à créer pour les différents concepts qui ne peuvent pas être construits à partir d'autres concepts. Dans ce cas, il s'agit des types *PLACE* et *TRANSITION* :

$$[PLACE, TRANSITION]$$

Les arcs sont construits à partir des types précédents. Le poids et le marquage initial sont construits à partir des types précédents et de l'ensemble des naturels $\mathbb{N}$.

Le réseau de Petri comprend :

1. $P = \{p_1, p_2, \ldots, p_n\}$ *un ensemble fini de places :*

   $$P : \mathbb{F}\, PLACE$$

   Cette définition permet de définir l'ensemble des places comme étant $P = \{p_1, p_2, \ldots, p_n\}$. L'ensemble des places peut aussi être $P = \{toto, bruno, \ldots, lolo\}$, ou même $P = \{p_1, transition, \ldots, sortir\}$ ...

2. $T = \{t_1, t_2, \ldots, t_n\}$ *un ensemble fini de transitions :*

   $$T : \mathbb{F}\, TRANSITION$$

3. $F \subseteq (P \times T) \cup (T \times P)$ *un ensemble d'arcs.*

   Il n'est pas possible de définir en Z un ensemble dont les individus ne sont pas tous du même type. Par contre, il est possible de construire un ensemble dont les individus sont construits à partir de plusieurs types. Il faut donc construire deux ensembles, un ensemble des arcs qui vont des places aux transitions ($arcPT$), et un ensemble des arcs qui vont des transitions vers les places ($arcTP$) :

   $$arcTP : TRANSITION \leftrightarrow PLACE$$
   $$arcPT : PLACE \leftrightarrow TRANSITION$$

   Il n'existe aucune contrainte sur les relations entre les individus de type *TRANSITION* et *PLACE*, sinon que les arcs ne doivent exister qu'entre des individus des

ensembles P et T. Pour la relation $arcPT$ (resp. $arcTP$), l'ensemble des individus de type $TRANSITION$ (resp. $PLACE$) représente le domaine (dom) de la relation. L'ensemble des individus de type $PLACE$ (resp. $TRANSITION$) représente l'image (ran) de la relation :

$$\begin{aligned}&\mathrm{dom}(arcTP) \subseteq T\\ &\mathrm{ran}(arcTP) \subseteq P\\ &\mathrm{dom}(arcPT) \subseteq P\\ &\mathrm{ran}(arcPT) \subseteq T\end{aligned}$$

Il est possible de se passer de la définition des ensembles P et T. En effet, nous pouvons considérer que seuls des arcs seront construits et manipulés. Cette possibilité permet d'avoir un schéma d'état plus court, plus lisible. Par contre, elle nous éloigne de la définition formelle de Murata. Cette possibilité sera tout de même étudiée dans l'annexe B.

4. $W : F \rightarrow \{1, 2, 3, \dots\}$ *est la fonction poids.*

Nous l'avons déjà dit, il n'est pas possible en Z de définir des ensembles dont tous les individus ne sont pas du même type. Étant donné que nous avons dû définir deux relations arc, nous devons définir deux fonctions poids. En Z, tout ensemble doit être typé, il n'est donc pas possible de définir directement la fonction $WarcTP$ (resp. $WarcPT$) par rapport à $arcTP$ (resp. $arcTP$). Elle doit être définie par rapport aux types $PLACE$, $TRANSITION$ et $\mathbb{N}_1$ ($\mathbb{N}$ moins 0).

$$\begin{aligned}&WarcTP : (TRANSITION \times PLACE) \nrightarrow \mathbb{N}_1\\ &WarcPT : (PLACE \times TRANSITION) \nrightarrow \mathbb{N}_1\end{aligned}$$

Pour spécifier que la fonction $WarcTP$ (resp. $WarcPT$) est la fonction poids associée à $arcTP$ (resp. $arcPT$) nous devons contraindre le domaine de chacune de ces fonctions ($\mathrm{dom}(WarcTP)$ de type $TRANSITION \times PLACE$ et $\mathrm{dom}(WarcPT)$ de type $PLACE \times TRANSITION$). Ces domaines doivent être identiques aux ensembles des arcs définis ($arcTP$ et $arcPT$), ce qui se traduit par :

$$\begin{aligned}&\mathrm{dom}(WarcTP) = arcTP\\ &\mathrm{dom}(WarcPT) = arcPT\end{aligned}$$

En fait, la notion de poids ne sera certainement jamais utilisée sans utiliser conjointement l'arc porteur de ce poids. Elle peut donc être intégrée à la définition des arcs. Afin de rester le plus proche possible de la définition de Murata, nous n'utilisons pas cette possibilité ici ... elle sera étudiée dans l'annexe B.

5. $M_0 : P \rightarrow \{1, 2, 3, \dots\}$ *est le marquage initial.*

Murata considère ici que le marquage initial d'une place ne peut pas prendre la valeur 0. Si nous respectons cette définition, certaines places auraient un marquage initial, d'autres pas. Il en est de même pour le marquage à chaque évolution. Examinons rapidement la conséquence de cette modélisation sur le comportement dynamique du modèle (que nous examinerons plus en détail plus tard). Supposons le tir d'une transition reliée par un arc de poids 1 à une place en amont comportant 1 jeton. Le tir de la transition retire 1 jeton de cette place amont (poids de l'arc). Si nous respectons la définition proposée, le résultat est donc : « plus de marquage » ! Puis lorsque un jeton sera rajouté à cette place lors du tir d'une autre transition, le marquage passera de « pas de marquage » à « 1 jeton » ... Nous considérons que cette modélisation n'est pas cohérente et préférons considérer qu'un marquage peut prendre la valeur 0, ce qui permet effectivement de calculer l'évolution du marquage. De plus ceci est plus fidèle à l'usage de l'utilisation des RdP : lorsqu'un RdP est traduit sous forme de matrice, les éléments correspondant aux places n'ayant pas de jeton ont pour valeur 0.

Le marquage initial est donc une fonction de type $PLACE$ (P est un ensemble de type $PLACE$) vers $\mathbb{N}$ (nous écrivons $M0$ plutôt que M_0 pour des problèmes d'utilisation de Z/EVES, logiciel de preuve sur Z).

$$M0 : PLACE \rightarrow \mathbb{N}$$

Un marquage ne peut être associé qu'à une place de l'ensemble P :

$$dom(M0) = P$$

6. $P \cap T = \emptyset$.

Ceci n'a pas besoin d'être spécifié en Z, et ne peut de toute façon pas être écrit, puisque les deux ensembles sont de deux types différents, donc ils n'ont pas d'intersection.

7. $P \cup T \neq \emptyset$.

Ceci signifie qu'il existe au moins un individu dans l'ensemble P ou dans l'ensemble T. Ceci ne peut pas être écrit en Z sous la forme d'une union car les deux ensembles sont de types différents. Par contre, nous pouvons écrire que la somme des cardinalités des ensembles P et T doit être supérieure ou égale à 1 (la cardinalité d'un ensemble quel qu'il soit est un individu de $\mathbb{N}$) :

$$\#P + \#T \geq 1$$

Une autre solution aurait été d'écrire que P est non vide ou T est non vide :

$$P \neq \emptyset \vee T \neq \emptyset$$

8. *Une structure de réseau de Petri N = (P, T, F, W) sans marquage initial spécifique est noté N.*
Un réseau de Petri avec un marquage initial est noté (N, M_0).

Dans la mesure où nous souhaitons construire un méta-modèle appréhendant à la fois les aspects statique et dynamique du RdP généralisé, il n'est pas souhaitable de distinguer ces deux aspects qui doivent tous les deux faire partie du schéma présentant l'état du RdP généralisé (ce schéma est nommé PN par analogie à PN = (P, T, F, W, M_o)) :

PN

$P : \mathbb{F}\, PLACE$
$T : \mathbb{F}\, TRANSITION$
$arcTP : TRANSITION \leftrightarrow PLACE$
$arcPT : PLACE \leftrightarrow TRANSITION$
$WarcTP : TRANSITION \times PLACE \pfun \mathbb{N}_1$
$WarcPT : PLACE \times TRANSITION \pfun \mathbb{N}_1$
$M0 : PLACE \pfun \mathbb{N}$

$\mathrm{dom}(arcTP) \subseteq T$
$\mathrm{ran}(arcTP) \subseteq P$
$\mathrm{dom}(arcPT) \subseteq P$
$\mathrm{ran}(arcPT) \subseteq T$
$\mathrm{dom}(WarcTP) = arcTP$
$\mathrm{dom}(WarcPT) = arcPT$
$\mathrm{dom}(M0) = P$
$\#P + \#T \geq 1$

9. *Un marquage (état) assigne à chaque place un entier non négatif. Si un marquage assigne à chaque place p un entier non négatif k, nous disons que p est marqué avec k jetons. Un marquage est noté M, un m-vecteur, où m est le nombre total de places. La* $p^{ième}$ *composante de M, notée M(p), est le nombre de jetons de la place p.*

Pour décrire l'évolution du marquage d'un RdP généralisé, il faut que ce marquage fasse partie de l'état du RdP généralisé, et donc du schéma PN. Deux possibilités existent, la première consiste à remplacer $M0$ par M, la seconde à conserver $M0$ et à rajouter M. La deuxième possibilité permet de conserver un sens particulier à $M0$, ce qui nous semble indispensable pour la vérification de certaines propriétés (les propriétés comportementales). C'est cette possibilité qui a été choisie.

Les places ont été définies comme étant un ensemble, qui est par définition non ordonné. De plus les places ne sont pas numérotées. Il n'est donc pas possible d'utiliser

une définition vectorielle du marquage puisque la p$^{\text{ième}}$ composante de ce vecteur ne pourrait être rattachée à aucune place en particulier. Nous utilisons pour M une définition similaire à celle de $M0$, une fonction de $PLACE$ vers $\mathbb{N}$ (à rajouter à PN) :

$$M : PLACE \rightarrow \mathbb{N}$$

Un marquage ne peut être associé qu'à une place de l'ensemble P :

$$dom(M) = P$$

10. *En modélisation, utilisant les concepts de conditions et d'événements, les places représentent les conditions, et les transitions représentent les événements. Une transition (un événement) a un certain nombre de places d'entrée et de sortie représentant respectivement les pré-conditions et les post-conditions de l'événement.*

    Le comportement de certains systèmes peut être décrit en termes d'états de systèmes et de leurs changements.

    Afin de simuler le comportement dynamique d'un système, un état ou marquage d'un réseau de Petri est changé suivant les règles de transition (tir) suivantes :

    1° *une transition t est dite validée si chaque place p d'entrée de t est marquée avec au moins w(p,t) jetons, où w(p,t) est le poids de l'arc de p vers t ;*

    2° *une transition validée doit ou ne doit pas être tirée (en fonction de l'événement qui arrive) ;*

    3° *le tir d'une transition validée t retire w(p,t) jetons de chaque place d'entrée p de t, et rajoute w(t,p) jetons à chaque place de sortie p de t, où w(t,p) est le poids de chaque arc de t vers p.*

    Une transition sans aucune place d'entrée est appelée une transition source, et une sans place de sortie est appelée transition puits. Une transition source est inconditionnellement validée, et le tir d'une transition puits consomme des jetons, mais n'en produit aucun.

    L'ensemble des transitions validées *enabled'* (après opération) est composé de deux ensembles :

    (a) l'ensemble des individus t, de type $TRANSITION$, pour lesquels quel que soit p de type $PLACE$, il existe un arc de p vers t, et le nombre de jetons de la place p soit supérieur ou égal au poids de l'arc de p vers t :

    $$\{t : TRANSITION \mid (\forall\, p : PLACE \bullet (p,t) \in arcPT' \wedge WarcPT'(p,t) < M'(p)) \bullet t\}$$

(b) l'ensemble des individus t, de type *TRANSITION*, faisant partie de l'ensemble T mais ne faisant pas partie de l'ensemble image de $arcTP$.

$$\{t : TRANSITION \mid t \in T \land t \notin \mathrm{ran}(arcPT') \bullet t\}$$

L'union de ces deux ensembles peut s'exprimer sous la forme :

$$enabled' = \{t : TRANSITION \mid (\forall p : PLACE \bullet \\ (p,t) \in arcPT' \land WarcPT'(p,t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\}$$

Cette expression de l'ensemble des transitions validées permet de déterminer, à tout instant, le contenu de cet ensemble, indépendamment des états précédents. Elle fera donc partie du schéma décrivant l'évolution du RdP généralisé, le schéma *Play* (un schéma décrivant une opération doit décrire les transformations de tous les ensembles de l'état). Il est donc inutile qu'elle fasse partie de l'état du RdP généralisé, le schéma PN (il y aurait redondance). Le schéma PN ne contient donc que les déclarations :

$$enabled : \mathbb{F}\, TRANSITION$$
$$enabled \subseteq T$$

La transition tirée *fired*? est choisie (par l'utilisateur) dans l'ensemble *enabled*.

Le calcul du marquage est différent suivant que les places sont reliées par un arc ($arcPT$ ou $arcTP$) à la transition *fired*? ou pas. Quatre cas doivent être envisagés :

- la place est reliée à la transition *fired?* par un arc de $arcPT$ et par un arc de $arcTP$. Le nouveau marquage est égal à l'ancien moins le poids de l'arc de $arcPT$ plus le poids de l'arc de $arcTP$:

$$p \in arcPT^{\sim}(\!|\, \{Fired?\} \,|\!) \cap arcTP(\!|\, \{Fired?\} \,|\!)$$
$$\land\ M'(p) = M(p) - WarcPT(p \mapsto Fired?) + WarcTP(Fired? \mapsto p)$$

- la place est reliée à la transition *fired?* par un arc de $arcPT$ mais pas par un arc de $arcTP$. Le nouveau marquage est égal à l'ancien moins le poids de l'arc de $arcPT$:

$$p \in arcPT^{\sim}(\!|\, \{Fired?\} \,|\!) \setminus arcTP(\!|\, \{Fired?\} \,|\!)$$
$$\land\ M'(p) = M(p) - WarcPT(p \mapsto Fired?)$$

- la place est reliée à la transition *fired?* par un arc de $arcTP$ mais pas par un arc de $arcPT$. Le nouveau marquage est égal à l'ancien plus le poids de l'arc de $arcTP$:

$$p \in arcTP(\!|\, \{Fired?\} \,|\!) \setminus arcPT^{\sim}(\!|\, \{Fired?\} \,|\!)$$
$$\land\ M'(p) = M(p) + WarcTP(Fired? \mapsto p)$$

– la place n'est reliée à la transition *fired?* par aucun arc. Le nouveau marquage est égal à l'ancien :

$$p \notin arcPT^{\sim}(\!|\ \{Fired?\}\ |\!) \cup arcTP(\!|\ \{Fired?\}\ |\!)$$
$$\land\ M'(p) = M(p)$$

Le schéma Play est donc :

Play

$$\Delta PN$$
$$Fired? : TRANSITION$$

$$usePN = play$$
$$Fired? \in enabled$$
$$P' = P$$
$$T' = T$$
$$arcTP' = arcTP$$
$$arcPT' = arcPT$$
$$WarcTP' = WarcTP$$
$$WarcPT' = WarcPT$$
$$M0' = M0$$
$$\forall p : PLACE \mid p \in P \bullet$$
$$(p \in arcPT^{\sim}(\!|\ \{Fired?\}\ |\!) \cap arcTP(\!|\ \{Fired?\}\ |\!)$$
$$\land\ M'(p) = M(p) - WarcPT(p \mapsto Fired?) + WarcTP(Fired? \mapsto p))$$
$$\lor$$
$$(p \in arcPT^{\sim}(\!|\ \{Fired?\}\ |\!) \setminus arcTP(\!|\ \{Fired?\}\ |\!)$$
$$\land\ M'(p) = M(p) - WarcPT(p \mapsto Fired?))$$
$$\lor$$
$$(p \in arcTP(\!|\ \{Fired?\}\ |\!) \setminus arcPT^{\sim}(\!|\ \{Fired?\}\ |\!)$$
$$\land\ M'(p) = M(p) + WarcTP(Fired? \mapsto p))$$
$$\lor$$
$$(p \notin arcPT^{\sim}(\!|\ \{Fired?\}\ |\!) \cup arcTP(\!|\ \{Fired?\}\ |\!)$$
$$\land\ M'(p) = M(p))$$
$$enabled' = \{t : TRANSITION \mid (\forall p : PLACE \bullet$$
$$(p,t) \in arcPT' \land\ WarcPT'(p,t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\}$$
$$usePN' = usePN$$

4.3 L'étude de la construction d'un modèle

La rigueur d'un méta-modèle du RdP généralisé passe par l'étude de la cohérence de l'ensemble des opérations manipulant l'état du RdP généralisé. La description des opérations de construction du RdP généralisé n'est pas décrite par Murata. Elles sont cependant à la base de la description de méthodes d'aide à la construction de réseaux de Petri. Nous avons donc voulu décrire ces opérations afin de valider les possibilités de notre démarche pour la description des méthodes de construction de modèles, de poser les bases de méthodes de construction de réseaux de Petri généralisés et de nous assurer de la cohérence du schéma *PN* vis à vis de ces opérations.

Tout d'abord, nous avons considéré qu'il n'était pas possible de modifier la structure

du RdP généralisé au cours du jeu. Nous avons donc créé un nouveau type $USE ::= build \mid play$ permettant, à travers la variable *use*, de définir si le RdP généralisé est en cours de construction ($use = build$) ou en cours de jeu ($use = play$).

Le schéma *PN* précise qu'un RdP généralisé est constitué d'au moins une place ou une transition ($\#P + \#T \geq 1$). Cette règle n'est pas applicable au début de la construction d'un RdP généralisé, à moins d'imposer une initialisation comprenant une place (ou une transition). Nous avons fait le choix contraire qui consiste à supprimer cette obligation. Par contre, celle-ci sera vérifiée avant de simuler le RdP généralisé (voir plus loin le schéma *StartPlay*).

Nous avons défini cinq schémas (*AddP*, *AddT*, *AddarcTP*, *AddarcPT* et *AddM0*) de construction du RdP généralisé. Chacun de ces schémas ne s'exécute qu'à la condition que $use = build$. Pendant les étapes de construction, nous avons considéré que les ensembles correspondants à une vision dynamique ne devaient pas être définis. Ils le sont pourtant car ils font partie de l'état du RdP généralisé. Seuls les ensembles M et *enabled* sont de ce type car ce sont les seuls ensembles de l'état du RdP généralisé modifiés en cours de simulation (ce n'est pas le cas notamment de $M0$). Cependant, ceci n'est pas applicable pour l'ensemble M puisque nous avons donné comme règle que chaque individu de l'ensemble P a un marquage, donc M ne peut pas être égal à l'ensemble vide. Nous avons fait le choix d'imposer $M = M0$ et $M' = M0'$ pendant les opérations de construction. Par contre l'ensemble *enabled* est considéré comme vide pendant la construction.

Le schéma *AddP* (resp. *AddT*) permet d'ajouter une place (resp. une transition) au RdP généralisé. Pour les deux opérations d'ajout d'une place et d'une transition, nous avons fait deux choix. Le premier choix a consisté à considérer qu'il n'est pas normal d'ajouter une place ou une transition à un réseau comportant déjà cette place ou cette transition. Nous avons considéré que si le concepteur tente cette opération, c'est qu'il se trompe, et donc que l'ajout ne doit pas être autorisé. Pour le schéma *AddP*, la nouvelle place (*NewPlace?*) ne doit donc pas déjà appartenir à l'ensemble P. Pour le schéma *AddT*, la nouvelle transition (*NewTransition?*) ne doit pas déjà appartenir à l'ensemble T. Le second choix a été de considérer que l'usage consiste à créer la structure d'un RdP généralisé, sans se soucier du marquage des places. Nous avons donc choisi de mettre comme marquage (M' et $M0'$) de la nouvelle place *NewPlace?*, dans le schéma *AddP*, la valeur 0.

AddP

ΔPN
$NewPlace? : PLACE$

$usePN = build$
$NewPlace? \notin P$
$P' = P \cup \{NewPlace?\}$
$T' = T$
$arcTP' = arcTP$
$arcPT = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0 \cup \{NewPlace? \mapsto 0\}$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddT

ΔPN
$NewTransition? : TRANSITION$

$usePN = build$
$NewTransition? \notin T$
$P' = P$
$T' = T \cup \{NewTransition?\}$
$arcTP' = arcTP$
$arcPT = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

Le schéma *AddarcTP* (resp. *AddarcPT*) permet d'ajouter un arc de type *arcTP* (resp. *arcPT*). Pour ces deux schémas, la place (*OutputPlace?* ou *InputPlace?*) et la transition (*Transition?*) doivent déjà appartenir respectivement à P et T. Ces schémas permettent également de donner la valeur du poids de l'arc créé. Ces opérations permettent également de modifier le poids d'un arc déjà existant, par utilisation de l'opérateur $\oplus$ dans la spécification de *WarcTP'* (schéma *AddarcTP*) et *WarcPT'* (schéma *WarcPT'* (schéma *AddarcPT*) (définition de l'opérateur $\oplus$ page 157). Dans le schéma *AddarcTP* par exemple, s'il existait déjà un arc *(Transition?, Outputplace?)* avec un poids, ce poids est modifié. Si nous avions utilisé l'opérateur $\cup$, il y aurait eu deux arcs *(Transition?, Outputplace?)* avec des poids différents.

AddarcTP

ΔPN
$OutputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$

$usePN = build$
$OutputPlace? \in P$
$Transition? \in T$
$P' = P$
$T' = T$
$arcTP' = arcTP \cup \{Transition? \mapsto OutputPlace?\}$
$arcPT' = arcPT$
$WarcTP' = WarcTP \oplus \{(Transition?, OutputPlace?) \mapsto Weight?\}$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddarcPT

ΔPN
$InputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$

$usePN = build$
$InputPlace? \in P$
$Transition? \in T$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT \cup \{InputPlace? \mapsto Transition?\}$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT \oplus \{(InputPlace?, Transition?) \mapsto Weight?\}$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

Le schéma *AddM0* permet de modifier le marquage initial d'une place (*NewMarquage?*). Cette place doit déjà appartenir à l'ensemble P. Comme pour les deux schémas précédents nous utilisons l'opérateur $\oplus$ plutôt que $\cup$ afin que cette opération serve bien à modifier le marquage d'une place et non pas à ajouter un deuxième marquage à une place (ce qui ne respecterait pas l'invariant).

AddM0

ΔPN
$Place? : PLACE$
$NewMarquage? : \mathbb{N}$

$usePN = build$
$Place? \in P$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0 \oplus \{Place? \mapsto NewMarquage?\}$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

L'introduction de la variable *usePN* oblige à créer deux schémas *StartPLay* et *StopPlay*, permettant de modifier la valeur de *usePN*.

Le schéma *StartPlay* permet également de déterminer un premier état de l'ensemble des transitions validées avant le premier tir, et de donner à la variable *usePN* la valeur *play*. Il permet également de vérifier la condition $\#P + \#T \geq 1$ avant la simulation.

StartPlay

ΔPN

$\#P + \#T \geq 1$
$usePN = build$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{t : TRANSITION \mid (\forall p : PLACE \bullet$
$\quad (p, t) \in arcPT' \land WarcPT'(p, t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\}$
$usePN' = play$

Le schéma *StopPlay* permet de redonner à la variable *usePN* sa valeur *build* : $usePN' = build$. Le RdP généralisé est donc à nouveau considéré comme étant en cours de construction, seules les opérations *AddP*, *AddT*, *AddarcPT*, *AddarcTP* et *AddM0* doivent être réalisées. Dans cet état, les ensembles M et *enabled* n'ont pas de sens mais doivent être définis (remarque déjà faite) : $M0'$ reste égal à $M0$ et M' est égal à $M0'$. La variable *usePN* retrouve sa valeur *build* : $usePN' = build$.

StopPlay

ΔPN

$usePN = play$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \emptyset$
$usePN' = build$

L'étude des opérations de construction d'un modèle ne peut être complète sans prévoir la description d'un état initial de ce modèle. Il est donc indispensable de prévoir un schéma, *PNinitial*, décrivant l'opération d'initialisation. Après initialisation, tous les ensembles sont vides, et la variable *usePN* a pour valeur *build*, afin de permettre les opérations de construction et d'interdire la simulation du RdP généralisé.

4.4 Méta-modèle complet

$[PLACE, TRANSITION]$

$USE ::= build \mid play$

PN

$P : \mathbb{F}\, PLACE$
$T : \mathbb{F}\, TRANSITION$
$arcTP : TRANSITION \leftrightarrow PLACE$
$arcPT : PLACE \leftrightarrow TRANSITION$
$WarcTP : TRANSITION \times PLACE \pfun \mathbb{N}_1$
$WarcPT : PLACE \times TRANSITION \pfun \mathbb{N}_1$
$M0 : PLACE \pfun \mathbb{N}$
$M : PLACE \pfun \mathbb{N}$
$enabled : \mathbb{F}\, TRANSITION$
$usePN : USE$

$\mathrm{dom}(arcTP) \subseteq T$
$\mathrm{ran}(arcTP) \subseteq P$
$\mathrm{dom}(arcPT) \subseteq P$
$\mathrm{ran}(arcPT) \subseteq T$
$\mathrm{dom}(WarcTP) = arcTP$
$\mathrm{dom}(WarcPT) = arcPT$
$\mathrm{dom}(M0) = P$
$\mathrm{dom}(M) = P$

PNinitial

PN'

$P' = \{\}$
$T' = \{\}$
$arcTP' = \{\}$
$arcPT' = \{\}$
$WarcTP' = \{\}$
$WarcPT' = \{\}$
$M0' = \{\}$
$M' = \{\}$
$enabled' = \{\}$
$usePN' = build$

AddP

ΔPN
$NewPlace? : PLACE$

$usePN = build$
$NewPlace? \notin P$
$P' = P \cup \{NewPlace?\}$
$T' = T$
$arcTP' = arcTP$
$arcPT = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0 \cup \{NewPlace? \mapsto 0\}$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddT

ΔPN
$NewTransition? : TRANSITION$

$usePN = build$
$NewTransition? \notin T$
$P' = P$
$T' = T \cup \{NewTransition?\}$
$arcTP' = arcTP$
$arcPT = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddarcTP

ΔPN
$OutputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$

$usePN = build$
$OutputPlace? \in P$
$Transition? \in T$
$P' = P$
$T' = T$
$arcTP' = arcTP \cup \{Transition? \mapsto OutputPlace?\}$
$arcPT' = arcPT$
$WarcTP' = WarcTP \oplus \{(Transition?, OutputPlace?) \mapsto Weight?\}$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddarcPT

ΔPN
$InputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$

$usePN = build$
$InputPlace? \in P$
$Transition? \in T$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT \cup \{InputPlace? \mapsto Transition?\}$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT \oplus \{(InputPlace?, Transition?) \mapsto Weight?\}$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddM0

ΔPN
$Place? : PLACE$
$NewMarquage? : \mathbb{N}$

$usePN = build$
$Place? \in P$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0 \oplus \{Place? \mapsto NewMarquage?\}$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

StartPlay

ΔPN

$\#P + \#T \geq 1$
$usePN = build$
$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{t : TRANSITION \mid (\forall\, p : PLACE \bullet$
$\quad (p, t) \in arcPT' \land WarcPT'(p, t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\}$
$usePN' = play$

Play

$$
\begin{array}{l}
\Delta PN \\
Fired? : TRANSITION \\
\hline
usePN = play \\
Fired? \in enabled \\
P' = P \\
T' = T \\
arcTP' = arcTP \\
arcPT' = arcPT \\
WarcTP' = WarcTP \\
WarcPT' = WarcPT \\
M0' = M0 \\
\forall p : PLACE \mid p \in P \bullet \\
\quad (p \in arcPT^{\sim}(\!| \{Fired?\} |\!) \cap arcTP(\!| \{Fired?\} |\!) \\
\quad \land M'(p) = M(p) - WarcPT(p \mapsto Fired?) + WarcTP(Fired? \mapsto p)) \\
\quad\quad \lor \\
\quad (p \in arcPT^{\sim}(\!| \{Fired?\} |\!) \setminus arcTP(\!| \{Fired?\} |\!) \\
\quad \land M'(p) = M(p) - WarcPT(p \mapsto Fired?)) \\
\quad\quad \lor \\
\quad (p \in arcTP(\!| \{Fired?\} |\!) \setminus arcPT^{\sim}(\!| \{Fired?\} |\!) \\
\quad \land M'(p) = M(p) + WarcTP(Fired? \mapsto p)) \\
\quad\quad \lor \\
\quad (p \notin arcPT^{\sim}(\!| \{Fired?\} |\!) \cup arcTP(\!| \{Fired?\} |\!) \\
\quad \land M'(p) = M(p)) \\
enabled' = \{t : TRANSITION \mid (\forall p : PLACE \bullet \\
\quad (p, t) \in arcPT' \land WarcPT'(p, t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\} \\
usePN' = usePN
\end{array}
$$

StopPlay

$$
\begin{array}{l}
\Delta PN \\
\hline
usePN = play \\
P' = P \\
T' = T \\
arcTP' = arcTP \\
arcPT' = arcPT \\
WarcTP' = WarcTP \\
WarcPT' = WarcPT \\
M0' = M0 \\
M' = M0' \\
enabled' = \{\} \\
usePN' = build
\end{array}
$$

4.5 Vérification de ce méta-modèle

Pour vérifier une spécification en langage Z, deux propriétés sont à vérifier :

- l'état spécifié ne comporte pas de contradiction. Ceci peut être établi en démontrant le théorème d'initialisation :

$$\exists\, PN' \bullet PNinitial$$

– les opérations sont totales, c'est à dire quelles sont toujours définies.

La première propriété est facilement prouvée :

Par définition

$$\exists\, PN' \bullet PNinitial$$

est équivalent à :

$$\begin{array}{ll}\exists & P' : \mathbb{F}\, PLACE \\ & T' : \mathbb{F}\, TRANSITION \\ & arcTP' : TRANSITION \leftrightarrow PLACE \\ & arcPT' : PLACE \leftrightarrow TRANSITION \\ & WarcTP' : TRANSITION \times PLACE \pfun \mathbb{N}_1 \\ & WarcPT' : PLACE \times TRANSITION \pfun \mathbb{N}_1 \\ & M0' : PLACE \pfun \mathbb{N} \\ & M' : PLACE \pfun \mathbb{N} \\ & enabled' : \mathbb{F}\, TRANSITION \\ & usePN' : USE \mid \\ & \qquad \mathrm{dom}(arcTP') \subseteq T' \land \\ & \qquad \mathrm{ran}(arcTP') \subseteq P' \land \\ & \qquad \mathrm{dom}(arcPT') \subseteq P' \land \\ & \qquad \mathrm{ran}(arcPT') \subseteq T' \land \\ & \qquad \mathrm{dom}(WarcTP') = arcTP' \land \\ & \qquad \mathrm{dom}(WarcPT') = arcPT' \land \\ & \qquad \mathrm{dom}(M0') = P' \land \\ & \qquad \mathrm{dom}(M') = P' \land \\ & \qquad P' = \{\} \land \\ & \qquad T' = \{\} \land \\ & \qquad arcTP' = \{\} \land \\ & \qquad arcPT' = \{\} \land \\ & \qquad WarcTP' = \{\} \land \\ & \qquad WarcPT' = \{\} \land \\ & \qquad M0' = \{\} \land \\ & \qquad M' = \{\} \land \\ & \qquad enabled' = \{\} \land \\ & \qquad usePN' = build \end{array}$$

Après simplification, ceci peut être déduit de l'ensemble des expressions suivantes :

$\mathrm{dom}\{\} \subseteq \{\}$
$\mathrm{ran}\{\} \subseteq \{\}$
$\mathrm{dom}\{\} = \{\}$
$\{\} \in \mathbb{F}\, PLACE$
$\{\} \in \mathbb{F}\, TRANSITION$
$\{\} \in TRANSITION \leftrightarrow PLACE$
$\{\} \in PLACE \leftrightarrow TRANSITION$
$\{\} \in TRANSITION \times PLACE \pfun \mathbb{N}_1$
$\{\} \in PLACE \times TRANSITION \pfun \mathbb{N}_1$
$\{\} \in PLACE \pfun \mathbb{N}$
$\{\} \in PLACE \pfun \mathbb{N}$
$build \in USE$

D'après les définitions du domaine et de l'image d'une relation, les trois premières expressions sont vraies. Les expressions suivantes sont aussi vraies par définition de l'ensemble vide. Quant à la dernière expression elle est vraie par définition du type *USE*.

Le calcul des préconditions des opérations donne:

pre $PNinitial \mathrel{\widehat=} true$
pre $AddP \mathrel{\widehat=} usePN = build \land NewPlace? \notin P$
pre $AddT \mathrel{\widehat=} usePN = build \land NewTransition? \notin T$
pre $AddarcTP \mathrel{\widehat=} usePN = build \land OutputPlace? \in P \land Transition? \in T$
pre $AddarcPT \mathrel{\widehat=} usePN = build \land InputPlace? \in P \land Transition? \in T$
pre $AddM0 \mathrel{\widehat=} usePN = build \land Place? \in P$
pre $StartPlay \mathrel{\widehat=} \#P + \#T \geq 1 \land usePN = build$
pre $Play \mathrel{\widehat=} usePN = play \land Fired? \in enabled$
pre $StopPlay \mathrel{\widehat=} usePN = play$

Aucune de ces opérations n'est totale (à part *PNintial*). Il est donc nécessaire de compléter ces schémas, et pour cela nous allons créer de nouveaux schémas. Par exemple, l'opération d'ajout d'une place va maintenant être décrite par le schéma *AddPlace* (voir plus loin la description exacte de ce schéma et des schémas utilisés). Ce schéma est la combinaison des schémas *AddP*, *Success*, *Pexists* et *PaddInPlay*. Les schémas *AddP*, *Pexists* et *PaddInPlay* ont chacun une précondition différente, ce qui permet de tenir compte de tous les cas. Nous avons, en plus, utilisé la variable $r!$ pour informer l'utilisateur sur la cause de l'échec de l'opération d'ajout, ou de son succès. Toutes les opérations sont ainsi redéfinies dans les pages suivantes, afin de rendre les opérations totales. Tous les schémas, ainsi que le calcul des préconditions sont présentés.

La nouvelle opération d'ajout d'une place se fait par le schéma *AddPlace* (le schéma *AddP* est le même que précédemment):

$Report ::=$
$\quad okay \mid place\_in\_use \mid place\_not\_in\_use$
$\quad \mid transition\_in\_use \mid transition\_not\_in\_use$
$\quad \mid bad\_operation \mid transition\_not\_in\_enabled \mid P\_and\_T\_empties$

Success

$r! : Report$

$r! = okay$

Pexists

ΞPN
$NewPlace? : PLACE$
$r! : Report$

$usePN = build$
$NewPlace? \in P$
$r! = place\_in\_use$

PaddInPlay

ΞPN
$NewPlace? : PLACE$
$r! : Report$

$usePN = play$
$r! = bad\_operation$

$$AddPlace \mathrel{\widehat=} (AddP \land Success) \lor Pexists \lor PaddInPlay$$

L'opération définie par *AddPlace* est totale :

$$\begin{aligned}
&\text{pre } AddPlace \\
&\Leftrightarrow (\text{pre } AddP \land \text{pre } Success) \lor \text{pre } Pexists \lor \text{pre } PaddInPlay \\
&\Leftrightarrow (usePN = build \land NewPlace? \notin P \land true) \lor \\
&\qquad (usePN = build \land NewPlace? \in P) \lor usePN = play \\
&\Leftrightarrow usePN = build \lor usePN = play \\
&\Leftrightarrow true
\end{aligned}$$

La nouvelle opération d'ajout d'une transition se fait par le schéma *AddTransition* (le schéma *AddT* est le même que précédemment) :

Texists

ΞPN
$NewTransition? : TRANSITION$
$r! : Report$

$usePN = build$
$NewTransition? \in T$
$r! = transition\_in\_use$

TaddInPlay

ΞPN
$NewTransition? : TRANSITION$
$r! : Report$

$usePN = play$
$r! = bad\_operation$

$AddTransition \mathrel{\widehat=} (AddT \land Success) \lor Texists \lor TaddInPlay$

L'opération définie par *AddTransition* est totale :

pre $AddTransition$
$\Leftrightarrow$ (pre $AddT \land$ pre $Success$) $\lor$ pre $Texists \lor$ pre $TaddInPlay$
$\Leftrightarrow (usePN = build \land NewTransition? \notin T \land true) \lor$
$\quad (usePN = build \land NewTransition? \in T) \lor usePN = play$
$\Leftrightarrow usePN = build \lor usePN = play$
$\Leftrightarrow true$

La nouvelle opération d'ajout d'un arc orienté transition vers place se fait par le schéma *AddArcTransitionPlace* :

PoutputNotExists

ΞPN
$OutputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$
$r! : Report$

$usePN = build$
$OutputPlace? \notin P$
$r! = place\_not\_in\_use$

ToutputNotExists

ΞPN
$OutputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$
$r! : Report$

$usePN = build$
$Transition? \notin T$
$r! = transition\_not\_in\_use$

AddarcTPInPlay

ΞPN
$OutputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$
$r! : Report$

$usePN = play$
$r! = bad\_operation$

$$AddArcTransitionPlace \mathrel{\widehat=} (AddarcTP \land Success) \lor PoutputNotExists \lor ToutputNotExists \lor AddarcTPInPlay$$

L'opération définie par *AddArcTransitionPlace* est totale :

$$\begin{aligned}
&\text{pre } AddArcTransitionPlace \\
&\Leftrightarrow (\text{pre } AddarcTP \land \text{pre } Success) \\
&\quad \lor \text{pre } PoutputNotExists \lor \text{pre } ToutputNotExists \lor \text{pre } AddarcTPInPlay \\
&\Leftrightarrow (usePN = build \land OutputPlace? \in P \land Transition? \in T \land true) \lor \\
&\quad (usePN = build \land OutputPlace? \notin P) \lor \\
&\quad (usePN = build \land Transition? \notin T) \lor usePN = play \\
&\Leftrightarrow usePN = build \lor usePN = play \\
&\Leftrightarrow true
\end{aligned}$$

La nouvelle opération d'ajout d'un arc orienté d'une place vers une transition se fait par le schéma *AddArcPlaceTransition* :

PinputNotExists

ΞPN
$InputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$
$r! : Report$

$usePN = build$
$InputPlace? \notin P$
$r! = place\_not\_in\_use$

TinputNotExists

ΞPN
$InputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$
$r! : Report$

$usePN = build$
$Transition? \notin T$
$r! = transition\_not\_in\_use$

AddarcPTinPlay

ΞPN
$InputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$
$r! : Report$

$usePN = play$
$r! = bad\_operation$

$$AddArcPlaceTransition \mathrel{\widehat=} (AddarcPT \land Success) \lor PinputNotExists \lor TinputNotExists \lor AddarcPTinPlay$$

L'opération définie par *AddArcPlaceTransition* est totale :

$$\begin{aligned}
&\text{pre } AddArcPlaceTransition \\
&\Leftrightarrow (\text{pre } AddarcPT \land \text{pre } Success) \\
&\qquad \lor \text{pre } PinputNotExists \lor \text{pre } TinputNotExists \lor \text{pre } AddarcPTinPlay \\
&\Leftrightarrow (usePN = build \land InputPlace? \in P \land Transition? \in T \land true) \\
&\qquad \lor (usePN = build \land InputPlace? \notin P) \\
&\qquad \lor (usePN = build \land Transition? \notin T) \lor usePN = play \\
&\Leftrightarrow usePN = build \lor usePN = play \\
&\Leftrightarrow true
\end{aligned}$$

La nouvelle opération de démarrage de la simulation se fait par le schéma *StartSimulation* :

OnePorOneT

ΞPN
$r! : Report$

$\#P + \#T = 0$
$usePN = build$
$r! = P\_and\_T\_empties$

StartPlayInPlay

ΞPN
$r! : Report$

$usePN = play$
$r! = bad\_operation$

$$StartSimulation \mathrel{\widehat=} (StartPlay \land Success) \lor OnePorOneT \lor StartPlayInPlay$$

L'opération définie par *StartSimulation* est totale :

$$
\begin{aligned}
&\text{pre } \mathit{StartSimulation} \\
&\Leftrightarrow (\text{pre } \mathit{StartPlay} \wedge \text{pre } \mathit{Success}) \vee \text{pre } \mathit{OnePorOneT} \vee \text{pre } \mathit{StartPlayInPlay} \\
&\Leftrightarrow (\#P + \#T \geq 1 \wedge \mathit{usePN} = \mathit{build} \wedge \mathit{true}) \vee (\#P + \#T = 0 \wedge \mathit{usePN} = \mathit{build}) \vee \mathit{usePN} = \mathit{play} \\
&\Leftrightarrow \mathit{usePN} = \mathit{build} \wedge (\#P + \#T \geq 1 \vee \#P + \#T = 0) \vee \mathit{usePN} = \mathit{play} \\
&\Leftrightarrow \mathit{usePN} = \mathit{build} \vee \mathit{usePN} = \mathit{play} \\
&\Leftrightarrow \mathit{true}
\end{aligned}
$$

La nouvelle opération de simulation se fait par le schéma *Simulation* :

ErrorFired

$$
\begin{aligned}
&\Xi PN \\
&\mathit{Fired}? : \mathit{TRANSITION} \\
&r! : \mathit{Report} \\
&\hline
&\mathit{usePN} = \mathit{play} \\
&\mathit{Fired}? \notin \mathit{enabled} \\
&r! = \mathit{transition\_not\_in\_enabled}
\end{aligned}
$$

PlayOrStopPlayInBuild

$$
\begin{aligned}
&\Xi PN \\
&r! : \mathit{Report} \\
&\hline
&\mathit{usePN} = \mathit{build} \\
&r! = \mathit{bad\_operation}
\end{aligned}
$$

$$\mathit{Simulation} \mathrel{\widehat{=}} (\mathit{play} \wedge \mathit{Success}) \vee \mathit{ErrorFired} \vee \mathit{PlayOrStopPlayInBuild}$$

L'opération définie par *Simulation* est totale :

$$
\begin{aligned}
&\text{pre } \mathit{Simulation} \\
&\Leftrightarrow (\text{pre } \mathit{play} \wedge \text{pre } \mathit{Success}) \\
&\qquad \vee \text{pre } \mathit{ErrorFired} \vee \text{pre } \mathit{StartPlayOrPlayInPlay} \\
&\Leftrightarrow (\mathit{usePN} = \mathit{play} \wedge \mathit{Fired}? \in \mathit{enabled} \text{pre } \mathit{true}) \vee \\
&\qquad (\mathit{usePN} = \mathit{play} \wedge \mathit{Fired}? \notin \mathit{enabled}) \vee \mathit{usePN} = \mathit{build} \\
&\Leftrightarrow \mathit{usePN} = \mathit{play} \vee \mathit{usePN} = \mathit{build} \\
&\Leftrightarrow \mathit{true}
\end{aligned}
$$

La nouvelle opération d'arrêt de la simulation se fait par le schéma *StopSimulation* :

$$\mathit{StopSimulation} \mathrel{\widehat{=}} (\mathit{StopPlay} \wedge \mathit{Success}) \vee \mathit{PlayOrStopPlayInBuild}$$

L'opération décrite par *StopSimulation* est totale :

$$
\begin{aligned}
&\text{pre } \mathit{StopSimulation} \\
&\Leftrightarrow (\text{pre } \mathit{StopPlay} \wedge \text{pre } \mathit{Success}) \vee \text{pre } \mathit{PlayOrStopPlayInBuild} \\
&\Leftrightarrow (\mathit{usePN} = \mathit{play} \wedge \mathit{true}) \vee \mathit{usePN} = \mathit{build} \\
&\Leftrightarrow \mathit{true}
\end{aligned}
$$

Nous avons donc vérifié et complété notre méta-modèle de telle façon qu'il ne comporte pas de contradiction et que toutes les opérations soient totales, c'est-à-dire que leur

précondition soit vraie. Ce travail indispensable peut être long mais ne présente pas de difficulté particulière.

4.6 Validation de ce méta-modèle

Pour la validation d'un méta-modèle en langage Z, deux approches peuvent être envisagées :

- d'une part, l'instanciation du méta-modèle, ce qui revient, dans notre cas, à tester les opérations de construction puis celles de simulation d'un RdP généralisé ;
- d'autre part, à vérifier que certaines propriétés attendues des réseaux de Petri sont vérifiées par les RdP généralisés construits à partir de notre méta-modèle.

En ce qui concerne le premier point, il est à préciser qu'une spécification en langage Z n'est pas forcément simulable. Ceci est le cas notamment des spécifications écrites exclusivement à partir de définitions axiomatiques. Nous nous sommes astreints à utiliser plutôt une description par état et opérations sur cet état. Cette démarche nous permet d'obtenir un méta-modèle simulable. Pour tester nos opérations, nous avons utilisé le logiciel Z-EVES [Saaltink95] [Saaltink97].

4.6.1 Instanciation du méta-modèle

Pour chacune des opérations, nous avons cherché à nous assurer que le comportement spécifié est bien le comportement souhaité. Pour que les opérations de construction puissent être utilisées dans des conditions favorables, elles doivent être utilisées dans un certain ordre. Par exemple, la première opération à effectuer est l'initialisation, l'ajout d'un arc ne peut se faire que s'il existe une place et une transition, etc. Nous avons donc dû écrire des schémas permettant de décrire la séquentialité des opérations. Par exemple, pour valider l'opération d'ajout d'un arc allant d'une transition à une place, nous avons écrit le schéma :

$$Test3 \mathrel{\widehat=} PNinitial \fatsemi AddP \fatsemi AddT \fatsemi AddarcTP$$

Nous avons testé ce schéma avec les valeurs :

$NewPlace? := p1$
$NewTransition? := t1$
$OutputPlace? := p1, Transition? := t1, Weight? := 1$

Le résultat obtenu a bien été :

$p1 \in PLACE$
$\wedge\ M0' = \{(p1, 0)\}$
$\wedge\ M' = \{(p1, 0)\}$
$\wedge\ t1 \in TRANSITION$
$\wedge\ \{p1\} = P'$
$\wedge\ T' = \{t1\}$
$\wedge\ WarcPT' = \{\}$
$\wedge\ WarcTP' = \{((t1, p1), 1)\}$
$\wedge\ \{\} = arcPT'$
$\wedge\ \{(t1, p1)\} = arcTP'$
$\wedge\ enabled' = \{\}$
$\wedge\ usePN' = build$

Pour les opérations de jeu du réseau, nous avons jugé nécessaire de disposer d'un modèle RdP généralisé comportant au moins une transition validée, et quatre places, chacune correspondant à un cas différent de calcul de marquage. Ce modèle RdP généralisé minimum est présenté figure 4.1.

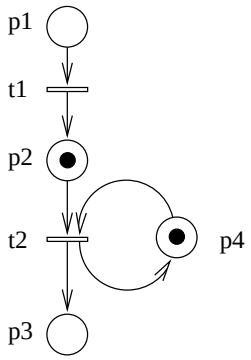


FIG. 4.1: *Exemple utilisé de modèle RdP généralisé*

Afin de simplifier la validation, nous avons modifié l'état initial afin qu'il décrive directement le RdP présenté :

PNinitial

PN'
$t1, t2 : TRANSITION$
$p1, p2, p3, p4 : PLACE$

$P' = \{p1, p2, p3, p4\}$
$T' = \{t1, t2\}$
$arcTP' = \{(t1, p2), (t2, p3), (t2, p4)\}$
$arcPT' = \{(p1, t1), (p2, t2), (p4, t2)\}$
$WarcTP' = \{((t1, p2), 1), ((t2, p3), 1), ((t2, p4), 1)\}$
$WarcPT' = \{((p1, t1), 1), ((p2, t2), 1), ((p4, t2), 1)\}$
$M0' = \{(p1, 0), (p2, 1), (p3, 0), (p4, 1)\}$
$M' = \{(p1, 0), (p2, 1), (p3, 0), (p4, 1)\}$
$enabled' = \{\}$
$usePN' = build$

Nous avons testé le schéma *Play* en créant un nouveau schéma :

$Test2 \mathrel{\widehat=} PNinitial \fatsemi StartPlay \fatsemi Play$

Nous avons testé ce schéma avec la valeur :

$Fired? := t2$

Le résultat obtenu a bien été le résultat attendu :

- après exécution de *StartPlay* nous avons $enabled' = \{t2\}$;
- après exécution de *Play* nous avons :
  - $enabled' = \{\}$ et
  - $M' = \{(p1, 0), (p2, 0), (p3, 1), (p4, 1)\}$.

Ce petit exemple, entre autres, nous a permis de valider notre méta-modèle par rapport à notre besoin, et ceci par la simple instanciation des ensembles définis.

4.6.2 Vérification de propriétés de modèles

Murata définit un certain nombre de propriétés vérifiables sur un RdP généralisé. Les propriétés comportementales définies sont l'atteignabilité, le caractère borné, la vivacité, la réversibilité, la persistance et la distance synchronique. Les propriétés structurelles sont la vivacité structurelle, la contrôlabilité, le caractère structurellement borné, le conservatisme, la répétitivité et la consistance.

Toutes les propriétés comportementales nécessitent, pour être vérifiées, la définition de séquences de tir à partir du marquage initial. Une séquence est décrite par $\sigma = M_0\ t_1\ M_1\ t_2\ M_2\ ...\ t_n\ M_n$ ou plus simplement par $\sigma = t_1\ t_2\ ...\ t_n$. A partir de cette séquence, il est possible de définir l'ensemble des marquages atteignables à partir de M_0 : $R(N, M_0)$ (ou plus simplement $R(M_0)$). Ces définitions ne font pas partie de la définition formelle proposée par Murata. La vérification de ces propriétés n'est donc pas directement exprimable à partir de notre méta-modèle. Elles nécessiteraient la définition de nouveaux ensembles et de propriétés exprimées à partir de ces ensembles. Afin de ne pas surcharger ce document, nous ne les exprimerons donc pas ici. Cependant nous avions précédemment démontré des preuves de propriétés du méta-modèle. La démarche est la même pour des propriétés d'instances de méta-modèles. Nous avons donc démontré que le langage Z est tout à fait apte à permettre la validation de méta-modèles exprimés en Z.

4.7 Conclusion

Le réseau de Petri généralisé est un langage difficile à méta-modéliser. En effet un méta-modèle abordant tous les aspects de ce langage doit traiter la syntaxe et la sémantique du langage, l'aspect statique des modèles produits mais aussi leur comportement dynamique et leur construction. Dans ce chapitre, nous avons construit un méta-modèle du RdP généralisé qui nous a effectivement permis de spécifier tous ces aspects et de démontrer les erreurs et les manques de la définition de référence. Ceci a été possible grâce à la rigueur de notre approche et du langage Z qui nous a, en outre, permis de vérifier et de valider notre méta-modèle sans avoir à le traduire dans un autre langage, donc sans perte de sémantique. Cet exemple difficile a été traité avec succès et a donc validé notre approche de méta-modélisation formelle. De par l'éventail des aspects méta-modélisés, cet exemple a également démontré la supériorité de notre approche sur celles basées sur la seule représentation de la structure des données. La rigueur et l'intégration du méta-modèle ont enfin montré tout l'intérêt du langage Z par rapport à l'utilisation conjointe de plusieurs langages semi-formels pour traiter des aspects complémentaires des méta-modèles.

Dans le chapitre cinq, nous allons maintenant valider notre approche sur une méthode intégrant plusieurs langages. De plus, cette méthode intégrera une partie discrète et une partie continue afin de montrer que notre approche peut être étendue aux systèmes hybrides.

Chapitre 5

Etude d'une méthode

L'objectif de ce chapitre est de valider notre approche sur un exemple de méthode. Nous avons pour cela choisi une méthode multi-langages que les approches existantes de méta-modélisation ne permettent pas d'étudier sous tous ses aspects. Notre choix s'est porté sur une méthode dans laquelle l'intégration se fait par le comportement dynamique du modèle. De plus, cette méthode présentée dans [Andreu et al.96] et [Andreu et al.98] est une méthode de modélisation de systèmes hybrides, c'est-à-dire comportant une partie discrète et une partie continue. Cet exemple nous permettra donc de montrer que notre approche n'est pas restreinte aux systèmes à événements discrets, mais peut être étendue aux systèmes hybrides.

Cette méthode rigoureuse est très clairement définie dans [Andreu96]. Elle utilise des modules décrits en réseaux de Petri et d'autres par des équations différentielles. Elle est assez flexible dans la mesure où différentes classes de RdP peuvent être utlisées. Elle est actuelle, des travaux complémentaires sont en cours [Champagnat98]. De plus, des exemples de l'utilisation de cette méthode sont disponibles. Toutes ces raisons nous ont amené à penser que cette méthode convient tout à fait pour l'utilisation qui est la notre ici : la méta-modélisation de l'intégration de différents langages hétérogènes et de leur utilisation conjointe dans une approche structurée.

Comme pour le chapitre précédent, nous présenterons d'abord une définition de référence, avant d'expliquer notre démarche de construction du méta-modèle et de présenter le méta-modèle final obtenu. La présentation de la définition sera structurée en trois parties. Tout d'abord, des extraits de [Andreu96], puis un exemple d'application et enfin des limitations apportées à la méthode.

5.1 Présentation des travaux

5.1.1 Objectif des travaux

L'objectif des travaux de David Andreu est d'avoir une approche globale, hiérarchisée et modulaire, des aspects temps-réel de la conduite des procédés hybrides, et notamment des procédés de traitement par lots. Le modèle du système, constitué du procédé et de la commande, résulte de l'utilisation de deux langages : les réseaux de Petri et les équations algébro-différentielles. Les réseaux de Petri sont utilisés pour aborder l'aspect discret, et les équations algébro-différentielles pour l'aspect continu.

Ce modèle est structuré en trois parties (figure 5.1, page 123). Une partie superviseur traduit la recette en consignes pour les régulateurs continus et en événements pour les automates programmables. Le superviseur est aussi chargé de la surveillance du procédé. Une partie comprend les contrôleurs locaux (régulateurs et automates), en boucle fermée avec le procédé. Cette partie génère les commandes du procédé. Une partie générateur d'événements observe le système et transmet au superviseur l'ensemble des événements détectés.

Le modèle pour la supervision est en fait constitué d'un modèle de la commande (discret) et d'un modèle hybride du procédé, appelé « modèle de référence hybride ». Ce dernier est utilisé essentiellement dans le cadre de la surveillance. Piloté par le modèle de la commande, il permet la détection de déviations du procédé (comparaison avec les événements détectés par le générateur d'événements) ainsi que la vérification de la faisabilité de l'ordonnancement établi (par simulation).

Le contrôle, la partie discrète du modèle de référence et les contrôleurs locaux discrets sont modélisés par des réseaux de Petri. La partie continue du modèle de référence, les contrôleurs locaux continus, le générateur d'événements sont modélisés par des équations algébro-différentielles.

5.1.2 Présentation détaillée

Relations entre les trois parties

L'entrée du superviseur est un plan de fabrication Σ_L, c'est-à-dire une suite d'ordres de lancement $l \in \Sigma_L$ de lots avec leurs dates et les équipements requis. Les recettes de contrôle sont traduites sous la forme de deux types de messages, U_c, $U_d \in \Sigma_U$, qui représentent l'ensemble des requêtes envoyées par le superviseur aux organes de commande. Les messages U_c sont des consignes envoyées aux régulateurs continus, et les messages U_d sont des événements qui provoquent des changements d'états des automates programmables industriels. Ces automates, connectés au procédé, reconfigurent physiquement l'installation

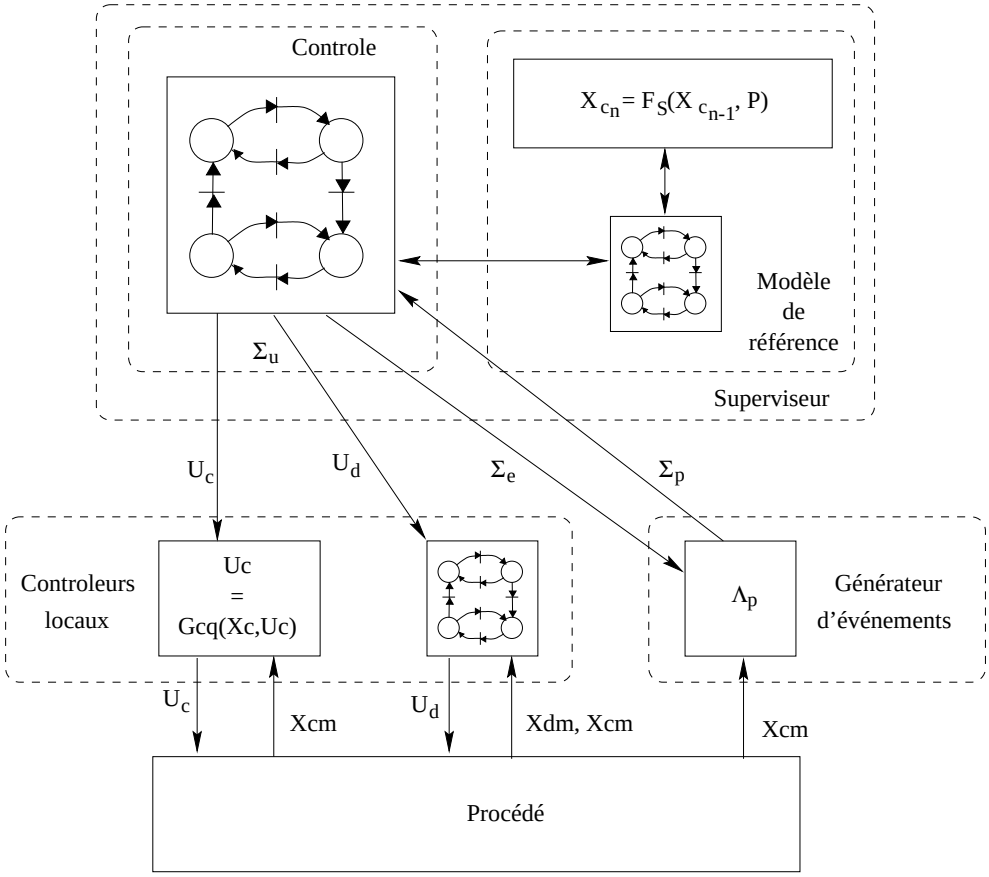


FIG. 5.1: *Supervision basée sur un modèle hybride [Andreu96]*

lors du changement de configuration par des ouvertures et des fermetures de vannes. Les régulateurs continus et les automates programmables industriels fonctionnent en boucle fermée.

Les régulateurs continus ont accès aux variables d'état continues x_{cm} mesurées par des capteurs. Ils reconstituent l'état continu, $x_c = F_c(x_{cm})$, et génèrent périodiquement les commandes continues : $u_c = G_{cq}(x_c, U_c)$.

Les automates programmables industriels détectent les variations de variables d'état discrètes mesurées x_{dm} et les franchissements de seuils par les variables d'état continues x_{cm}. Par un ensemble de fonctions logiques combinatoires (B), ils déterminent l'occurrence d'événements $e = B(x_{d_{n-1}}, x_{dm}, x_{cm})$, $x_{d_{n-1}}$ étant l'état discret précédent. A partir des événements, l'état discret suivant est obtenu, $x_{d_n} = F_d(x_{d_{n-1}}, e)$, et les commandes discrètes sont déduites et appliquées au cycle suivant : $u_d = G_d(x_{d_n})$. Une requête du superviseur (message U_d) peut ainsi donner lieu à l'application de plusieurs commandes discrètes : la requête est affinée.

La supervision n'a pas les mêmes contraintes temps réel strict que la commande locale. Elle n'accède donc pas directement aux variables d'état continues du procédé. La fonction

générateur d'événements est chargée de détecter les événements d'état provoquant un changement de phase ou de configuration tels que les franchissements de seuil. A chaque changement de configuration (c'est-à-dire à chaque changement d'état discret X_d pour le modèle utilisé par la supervision), le générateur d'événements doit être reprogrammé. En se basant sur le réseau de Petri décrivant la recette, cette reprogrammation est systématique. En effet, à partir de l'état discret courant X_d, c'est-à-dire du marquage du réseau de Petri qui indique la configuration active, nous pouvons déduire l'ensemble des événements Σ_e à observer par le générateur d'événements (Λ_P). Σ_e correspond à l'ensemble des événements à observer, et Σ_p a pour signification l'ensemble des événements détectés.

La partie supervision

La supervision doit assurer le bon fonctionnement du procédé, en le surveillant. La surveillance repose sur la détection des déviations par comparaison entre le comportement attendu et le comportement effectif. Cette comparaison repose donc sur la confrontation des observations issues du générateur d'événements, avec les résultats de la simulation. La simulation est fondée sur le « modèle de référence » qui, à son image, est hybride. La résolution des équations est supposée ne pas poser de problème.

Pour chaque transition de seuil sensibilisée par l'état discret courant, le système d'équation algébro-différentielle est utilisé pour évaluer la date au plus tôt ($d1$) et la date au plus tard ($d2$) de franchissement de seuil. Chaque fois qu'un événement d'état est détecté par le générateur d'événements, il est transmis à la supervision qui vérifie si sa date d'occurrence est compatible avec la fenêtre temporelle $[d1, d2]$. Si c'est le cas, l'état discret suivant est calculé, $X_{d_n} = F_d(X_{d_{n-1}}, P)$, le nouveau système d'équations algébro-différentielles est généré par l'évolution du marquage, et le générateur d'événements est reprogrammé. Si l'occurrence de l'événement se produit en dehors de la fenêtre temporelle, la transition correspondante n'est pas franchissable, et l'état discret suivant ne peut pas être atteint.

Les réseaux de Petri utilisés

La classe de réseau de Petri utilisée dans chaque partie dépend des besoins de modélisation : prise en compte explicite de l'aspect temporel, manipulation de structures de données, etc. Ainsi les réseaux de Petri utilisés pour modéliser les recettes sont les réseaux de Petri temporels à objets. La structure du réseau correspond à la procédure de la recette et les objets manipulés sont d'une part les objets « lot », et d'autre part les objets « ressource ». Les réseaux de Petri temporels à objets sont également utilisés pour la description des fonctions. Au plus bas niveau, la classe de réseaux de Petri utilisée est le réseau de Petri interprété de commande ou le grafcet.

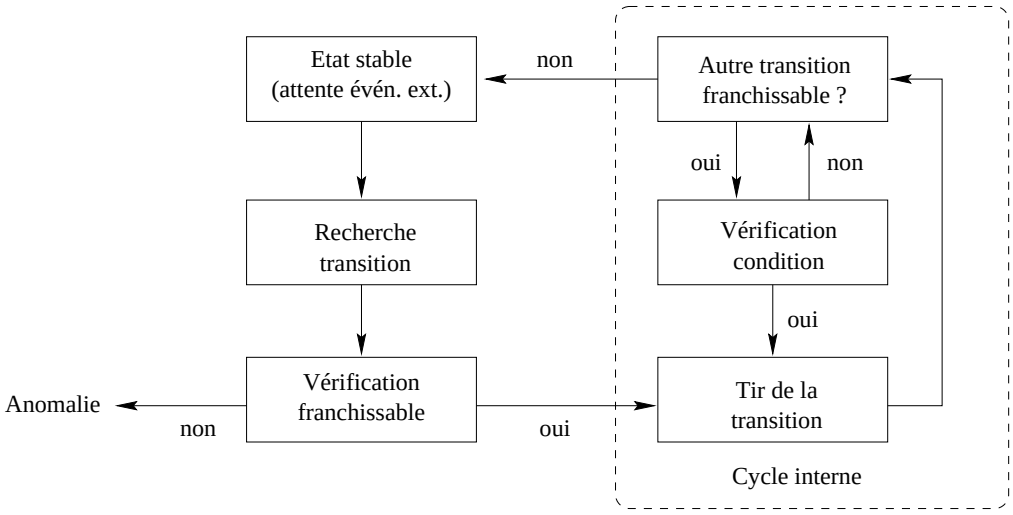


FIG. 5.2: *Principe du joueur de réseau de Petri [Andreu96]*

Le joueur de réseau de Petri utilisé est décrit par la figure 5.2. L'état stable d'un réseau est un marquage pour lequel seules des transitions associées à des événements externes sont franchissables. Le joueur se place donc en attente de l'occurrence d'un événement externe. Quand un événement externe se produit, le joueur cherche la transition à laquelle il est associé, et teste si elle est franchissable. Si la transition est franchissable, il la franchit, ainsi que toutes celles rendues franchissables par le nouveau marquage, jusqu'à atteindre un état stable. Par contre, si la transition ne peut pas être tirée, le joueur détecte une anomalie et reste dans l'état stable.

5.1.3 Un exemple

Cette méthode est relativement complexe. Aussi, afin de la rendre plus claire, nous en présentons un exemple d'application. De plus, cet exemple servira de support pour la validation de notre méta-modèle.

Présentation

L'exemple présenté ici, extrait de [Andreu96], traite d'une unité de stockage (figure 5.3, page 126). Cet exemple ayant pour objectif d'illustrer le concept de modèle de référence, il n'inclut pas les parties contrôleurs locaux et générateur d'événements.

L'unité de stockage est constituée d'un réservoir R pour lequel la variable d'intérêt est sa rétention massique M_R. En considérant le réservoir comme isotherme et parfaitement agité, le bilan de conservation de la matière s'écrit :

$$\text{vitesse d'accumulation - débit d'entrée + débit de sortie} = 0$$

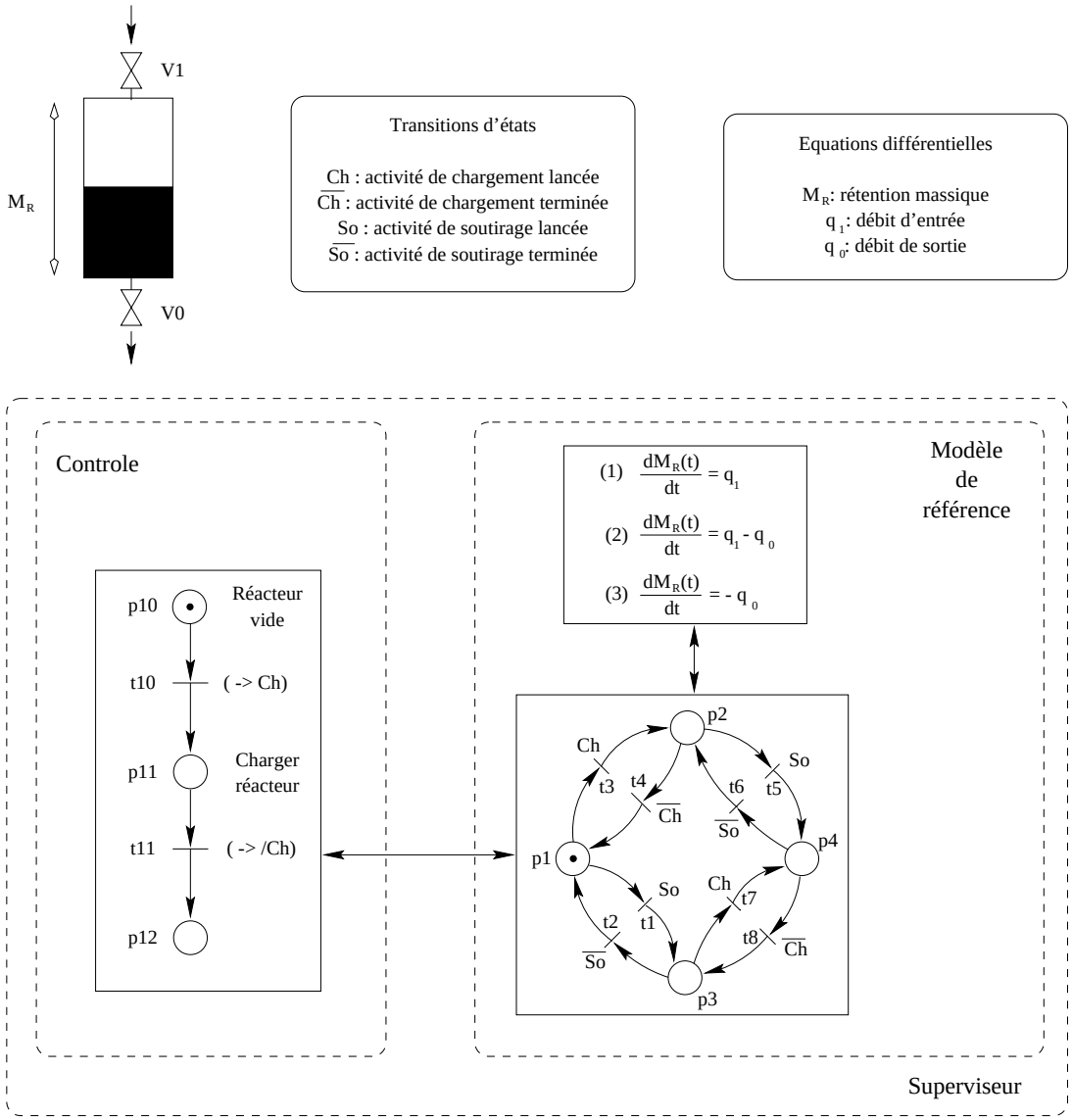


FIG. 5.3: *Exemple sur une unité de stockage [Andreu96]*

Le modèle de référence discret du réservoir est composé de quatres états discrets, les quatres places du réseau de Petri. Ces états correspondent aux configurations suivantes : « réservoir inactif » (place $p1$), « réservoir en remplissage » (place $p2$), « réservoir en soutirage » (place $p3$), et « réservoir simultanément en remplissage et soutirage » (place $p4$).

Ces quatre états discrets sont les configurations accessibles du réservoir. A chacun d'entre eux est associé le modèle comportemental selon une structure dépendant de l'état. Seul l'état « réservoir inactif » (vide) n'a pas d'équation associée, aucun phénomène continu n'ayant lieu dans cette configuration. Dans l'état « réservoir en remplissage »

(place $p2$), par exemple, le modèle se limite à :

$$\text{vitesse d'accumulation - débit d'entrée} = 0$$

C'est ce qui est exprimé par l'équation (1) associée à la place $p2$.

Le modèle de commande discret décrit le déroulement de la recette. L'exécution de cette recette est décomposée en trois étapes, chacune correspondant à une place du RdP : le lancement de l'opération (réacteur vide, place $p10$), le déroulement de l'opération (charger réacteur, place $p11$), et fin de l'opération (place $p12$).

Jeu du modèle

Avant le lancement de l'opération, la recette est dans l'état $p10$ et le modèle de référence du réacteur dans l'état discret $p1$. Les transitions sensibilisées sont la transition $t10$ sur le modèle de commande, et les transitions $t1$ et $t3$ sur le modèle de référence. Le lancement de la phase de chargement du réacteur correspond au franchissement de la transition $t10$. Ce franchissement n'est possible que si la requête de remplissage du réacteur est autorisée par son modèle de référence, c'est-à-dire si la transition $t3$, qui est fusionnée avec la transition $t10$ (Ch), peut être franchie simultanément à la transition $t10$.

Le lancement effectué (tir des transitions $t1$ et $t10$), le générateur d'événements a été programmé par la recette (tir de $t10$), et le nouvel état atteint par les modèles discrets est traduit par le marquage des places $p11$ et $p2$. L'ensemble des transitions sensibilisées devient $t11$, $t4$ et $t5$. La place $p2$ étant marquée, le modèle de référence calcule la fenêtre temporelle de réception de l'événement d'état associée à la transition $t4$ ($[d1, d2]$), ainsi que la valeur attendue de la rétention massique du réacteur (valeur simulée).

Dès réception de l'événement « réacteur chargé », la transition $t11$ est franchie seulement si les deux conditions suivantes sont satisfaites :

- l'occurrence de l'événement a eu lieu dans l'intervalle de temps prévu : il s'agit de l'intervalle porté sur la transition $t4$. En effet, le tir de la transition $t11$ n'est possible que si la transition $t4$ avec laquelle elle est fusionnée est franchissable, or la transition $t4$ n'est franchissable que dans l'intervalle de temps $[d1, d2]$;

- l'écart entre la valeur mesurée de la rétention massique du réacteur, qui a été retournée par le générateur d'événements, et la valeur simulée est inférieure à un certain seuil. Dans ce cas, la composante continue du modèle de référence du réacteur est mise à jour : le modèle conserve ainsi une image fidèle de l'état effectif du procédé (état discret et état continu). L'état continu ainsi mis à jour constitue les valeurs initiales des prochains calculs.

L'évolution du modèle de la commande induit celle du modèle de référence : les modèles sont indépendants mais sont dynamiquement couplés lors de l'exécution de la recette.

Si l'événement se produit en dehors de la fenêtre temporelle déterminée, ou si l'écart entre les valeurs est supérieur au seuil spécifié, la transition t_4 n'est pas franchissable. Le modèle de référence ne peut donc pas évoluer, par conséquent le modèle de la commande est bloqué puisque la transition $t11$ n'est pas franchissable. Une défaillance a été détectée.

5.2 Construction du méta-modèle

5.2.1 Hypothèses de travail

Comme nous l'avons déjà dit, l'objectif de ce chapitre est de présenter l'intérêt de la méta-modélisation pour l'intégration de plusieurs langages au sein d'une méthode. Il est également souhaitable que cette méthode soit suffisamment simple pour être plus facilement appréhendable dans le cadre de cette présentation. Nous proposons donc de restreindre l'étude de la méthode à la seule étude de la supervision et l'utilisation de la méthode à des cas « simples » où la supervision ne comporte qu'une recette (un seul RdP de commande) sur un seul lot et qu'un seul modèle de référence (un seul RdP et un ensemble d'équations), soit une seule ressource. Cette hypothèse ne permet de représenter que des cas d'école, tel l'exemple précédent, mais ces cas sont suffisants du point de vue de notre problématique.

Ces restrictions nous permettent d'utiliser des RdP ne nécessitant pas l'extension objet. En effet, dans la méthode originale, les objets servent à définir les lots et les ressources utilisés. Or nous ne considérons qu'un seul lot et une seule ressource : il n'y a pas de besoin d'identification. Par contre, l'aspect temporel est indispensable et il est donc conservé.

5.2.2 Structuration du méta-modèle

Le méta-modèle d'une méthode multi-langages n'est pas simplement l'association des méta-modèles des langages utilisés dans cette méthode. Dans le chapitre précédent, nous avons présenté le méta-modèle du réseau de Petri généralisé, tel que défini par Murata [Murata89]. Les réseaux de Petri utilisés dans la méthode traitée ici sont des réseaux de Petri incluant en plus des événements et des fenêtres temporelles. Le méta-modèle de cette famille de réseau sera dérivé du RdP généralisé. Les méta-modèles de la commande et du modèle de référence discret seront de ce type de méta-modèle. Le modèle d'équation différentielle sera modélisé par un autre méta-modèle qui servira de type pour la définition des équations du modèle. Enfin l'intégration entre ces méta-modèles sera spécifiée par un autre schéma intégrant les méta-modèles précédents ainsi que les liens entre méta-modèles.

5.2.3 La modélisation du temps

Le modèle prend en compte le temps. Avant de construire la spécification des modèles, il est donc nécessaire de modéliser le temps. Dans la méthode, les équations différentielles sont des équations continues. Le temps est donc défini dans notre modèle comme étant continu : nous utilisons ici les extensions proposées par [Barden et al.94] et [Valentine95] qui nous permettent d'utiliser $\mathbb{R}$ comme type.

$$TIME == \mathbb{R}$$

L'écoulement du temps est représenté à travers la variable *date* et le schéma *Time* :

Time

$$date : TIME$$

La dynamique des modèles du système n'influe pas sur l'écoulement du temps. Nous ne modélisons donc pas cet écoulement. Le temps n'est modélisé que pour permettre le calcul de durées d'activation de places.

5.2.4 Les modèles discrets

Le méta-modèle

Le schéma définissant l'état du réseau de Petri est basé sur celui défini dans le chapitre précédent. Nous le rappelons ici :

$$[PLACE, TRANSITION]$$

$$USE ::= build \mid play$$

PN

$P : \mathbb{F}\, PLACE$
$T : \mathbb{F}\, TRANSITION$
$arcTP : TRANSITION \leftrightarrow PLACE$
$arcPT : PLACE \leftrightarrow TRANSITION$
$WarcTP : TRANSITION \times PLACE \pfun \mathbb{N}_1$
$WarcPT : PLACE \times TRANSITION \pfun \mathbb{N}_1$
$M0 : PLACE \pfun \mathbb{N}$
$M : PLACE \pfun \mathbb{N}$
$enabled : \mathbb{F}\, TRANSITION$
$usePN : USE$

$\mathrm{dom}(arcTP) \subseteq T$
$\mathrm{ran}(arcTP) \subseteq P$
$\mathrm{dom}(arcPT) \subseteq P$
$\mathrm{ran}(arcPT) \subseteq T$
$\mathrm{dom}(WarcTP) = arcTP$
$\mathrm{dom}(WarcPT) = arcPT$
$\mathrm{dom}(M0) = P$
$\mathrm{dom}(M) = P$

La famille de réseau de Petri utilisée dans la méthode est différente de celle que nous avons spécifiée dans le chapitre précédent. Ce réseau de Petri utilise en plus des événements, et des fenêtres temporelles sont associées aux transitions. Ce réseau de Petri est spécifié à travers le schéma PNTE (PN temporel à événements), qui étend le schéma PN avec les fenêtres temporelles et les événements. Ce schéma comprend également la variable *pndate* de type *TIME* qui permet de déterminer le temps d'activation d'une place avant occurrence d'un nouvel événement.

$[EVENT]$

PNTE

PN
$event : \mathbb{F}\, EVENT$
$window : TRANSITION \pfun TIME \times TIME$
$condition : TRANSITION \pfun EVENT$
$pndate : TIME$

$\mathrm{dom}(window) = T$
$\mathrm{dom}(condition) = T$
$\mathrm{ran}(condition) = event$

Le schéma précédent revient à écrire :

$$
\begin{array}{|l}
\hline
PNTE \\
\hline
P : \mathbb{F}\, PLACE \\
T : \mathbb{F}\, TRANSITION \\
arcTP : TRANSITION \leftrightarrow PLACE \\
arcPT : PLACE \leftrightarrow TRANSITION \\
WarcTP : TRANSITION \times PLACE \pfun \mathbb{N}_1 \\
WarcPT : PLACE \times TRANSITION \pfun \mathbb{N}_1 \\
M0 : PLACE \pfun \mathbb{N} \\
M : PLACE \pfun \mathbb{N} \\
enabled : \mathbb{F}\, TRANSITION \\
usePN : USE \\
event : \mathbb{F}\, EVENT \\
window : TRANSITION \pfun TIME \times TIME \\
condition : TRANSITION \pfun EVENT \\
pndate : TIME \\
\hline
\mathrm{dom}(arcTP) \subseteq T \\
\mathrm{ran}(arcTP) \subseteq P \\
\mathrm{dom}(arcPT) \subseteq P \\
\mathrm{ran}(arcPT) \subseteq T \\
\mathrm{dom}(WarcTP) = arcTP \\
\mathrm{dom}(WarcPT) = arcPT \\
\mathrm{dom}(M0) = P \\
\mathrm{dom}(M) = P \\
\mathrm{dom}(window) = T \\
\mathrm{dom}(condition) = T \\
\mathrm{ran}(condition) = event \\
\hline
\end{array}
$$

Les opérations de construction d'un modèle PNTE sont les mêmes que celles définies au chapitre précédent (pour les éléments communs aux deux familles de réseaux). Des opérations complémentaires de construction doivent être spécifiées, telles que l'ajout des événements et des fenêtres temporelles. Ces opérations sont identiques à celles déjà spécifiées, c'est pourquoi nous ne les présenterons pas. Par contre les opérations décrivant le joueur du PNTE sont différentes de celles du PN, car elles doivent tenir compte de l'aspect temporel du modèle.

Pour raccourcir l'écriture des schémas décrivant les opérations, nous introduisons un schéma intermédiaire $\Phi PNTE$. Le schéma $\Phi PNTE$ est égal au schéma $\Delta PNTE$, mais il impose en plus que la plupart des composants de $PNTE'$ soit égaux à ceux de $PNTE$. Pour ce faire, nous utilisons l'opérateur de masquage qui permet de supprimer des composants d'un schéma. Ainsi $\Xi PNTE \setminus (M, M', enabled, enabled', usePN, usePN', pndate, pndate')$ est égal au schéma $\Xi PNTE$, sans les composants M, M', $enabled$, $enabled'$, $usePN$, $usePN'$, $pndate$ et $pndate'$. Le schéma $\Phi PNTE$ permet donc de spécifier rapidement quels sont les ensembles et individus qui ne sont pas modifiés par les opérations :

$$\Phi PNTE \mathrel{\widehat=} \Delta PNTE \land (\Xi PNTE \setminus (M, M', enabled, enabled', usePN, usePN', pndate, pndate'))$$

Ce qui revient à écrire (les variables écrites en caractères gras sont celles qui ont

été masquées dans la déclaration précédente, et qui n'apparaissent pas dans la partie prédicative du schéma suivant) :

$\Phi PNTE$

$P, P' : \mathbb{F}\, PLACE$
$T, T' : \mathbb{F}\, TRANSITION$
$arcTP, arcTP' : TRANSITION \leftrightarrow PLACE$
$arcPT, arcPT' : PLACE \leftrightarrow TRANSITION$
$WarcTP, WarcTP' : TRANSITION \times PLACE \pfun \mathbb{N}_1$
$WarcPT, WarcPT' : PLACE \times TRANSITION \pfun \mathbb{N}_1$
$M0, M0' : PLACE \pfun \mathbb{N}$
$\mathbf{M}, \mathbf{M'} : \mathbf{PLACE} \pfun \mathbb{N}$
$\mathbf{enabled}, \mathbf{enabled'} : \mathbb{F}\, \mathbf{TRANSITION}$
$\mathbf{usePN}, \mathbf{usePN'} : \mathbf{USE}$
$event, event' : \mathbb{F}\, EVENT$
$window, window' : TRANSITION \pfun TIME \times TIME$
$condition, condition' : TRANSITION \pfun EVENT$
$\mathbf{pndate}, \mathbf{pndate'} : \mathbf{TIME}$

$P' = P$
$T' = T$
$arcTP' = arcTP$
$arcPT' = arcPT$
$WarcTP' = WarcTP$
$WarcPT' = WarcPT$
$M0' = M0$
$event' = event$
$window' = window$
$condition' = condition$

La déclaration de ce schéma dans les schémas décrivant des opérations sur *PNTE* permettra de n'avoir à spécifier que les ensembles transformés par celles-ci.

Le schéma *PNTE_StartPlay* est très proche du schéma *StartPlay*. La différence notable est l'initialisation de la variable *pndate* à la valeur *date*.

PNTE_StartPlay

$\Phi PNTE$
$\Xi Time$

$\#P + \#T \geq 1$
$usePN = build$
$M' = M0'$
$enabled' = \{t : TRANSITION \mid (\forall p : PLACE \bullet$
$\quad (p, t) \in arcPT' \land WarcPT'(p, t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\}$
$usePN' = play$
$pndate' = date$

Le schéma *PNTE_Play* spécifie l'évolution du marquage avec la même procédure que le schéma *Play*. Par contre, la transition *Fired* n'est pas la variable d'entrée. Elle est déduite à partir de l'entrée *Event?*, après vérification de certaines conditions. La variable *pndate* est réinitialisée par ce schéma.

PNTE_Play

$$
\begin{array}{l}
\Phi PNTE \\
\Xi Time \\
Event? : EVENT \\
\hline
Event? \in event \\
usePN = play \\
\exists\, Fired : TRANSITION \mid \\
\quad Fired \in enabled \land \\
\quad condition(Fired) = Event? \land \\
\quad ((Fired \in \mathrm{dom}\, window \land \\
\quad (\forall\, t1, t2 : TIME \mid (t1, t2) = window(Fired) \bullet \\
\qquad t1 < (date - pndate) \land \\
\qquad t2 > (date - pndate))) \lor \\
\quad Fired \notin \mathrm{dom}\, window) \bullet \\
\quad (\forall\, p : PLACE \mid p \in P \bullet \\
\qquad (p \in arcPT^{\sim}(\!|\, \{Fired\} \,|\!) \cap arcTP(\!|\, \{Fired\} \,|\!) \\
\qquad \land\, M'(p) = M(p) - WarcPT(p \mapsto Fired) + WarcTP(Fired \mapsto p)) \\
\qquad\quad \lor \\
\qquad (p \in arcPT^{\sim}(\!|\, \{Fired\} \,|\!) \setminus arcTP(\!|\, \{Fired\} \,|\!) \\
\qquad \land\, M'(p) = M(p) - WarcPT(p \mapsto Fired)) \\
\qquad\quad \lor \\
\qquad (p \in arcTP(\!|\, \{Fired\} \,|\!) \setminus arcPT^{\sim}(\!|\, \{Fired\} \,|\!) \\
\qquad \land\, M'(p) = M(p) + WarcTP(Fired \mapsto p)) \\
\qquad\quad \lor \\
\qquad (p \notin arcPT^{\sim}(\!|\, \{Fired\} \,|\!) \cup arcTP(\!|\, \{Fired\} \,|\!) \\
\qquad \land\, M'(p) = M(p))) \\
enabled' = \{t : TRANSITION \mid (\forall\, p : PLACE \bullet \\
\quad (p, t) \in arcPT' \land WarcPT'(p, t) < M'(p)) \lor t \notin \mathrm{ran}(arcPT') \bullet t\} \\
usePN' = usePN \\
pndate' = date
\end{array}
$$

Le schéma *PNTE_StopPlay*, comme le schéma *StopPlay*, a surtout comme objectif de remettre la variable *usePN* a la valeur *play*. Le schéma *Time* n'est pas déclaré car la variable *date* n'est pas utilisée.

PNTE_StopPlay

$$
\begin{array}{l}
\Phi PNTE \\
\hline
usePN = play \\
M' = M0' \\
enabled' = \{\} \\
usePN' = build \\
pndate' = 0
\end{array}
$$

Pour un nouveau besoin en modélisation, la nécessité d'un enrichissement du RdP s'est fait sentir : le méta-modèle du RdP temporel à événements (PNTE). La partie état du méta-modèle de ce PNTE a été obtenue en ajoutant des ensembles et des fonctions à celui de PN. Ces ajouts ont modifié le comportement dynamique des modèles, ce qui a entrainé une modification des opérations de jeu. Ces modifications ne sont pas seulement des ajouts des changements d'état des nouveaux ensembles. Des modifications dans les

changements d'état des ensembles contenus dans PN ont également eu lieu.

Un exemple

Cet exemple présente un réseau, dans son état après construction et affectation du marquage initial.

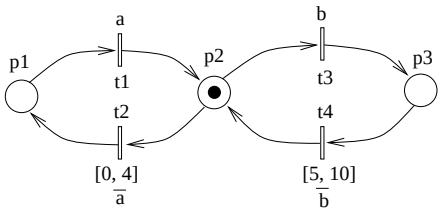


FIG. 5.4: *Exemple de rdP temporel à événements*

L'instanciation sur cet exemple du schéma *PNTE* après les opérations de construction et de marquage initial est représentée ici sous forme de schéma pour faciliter la lecture :

PNTE'

$P' = \{p1, p2, p3\}$
$T' = \{t1, t2, t3, t4\}$
$arcTP' = \{(t1, p2), (t2, p1), (t3, p3), (t4, p2)\}$
$arcPT' = \{(p1, t1), (p2, t3), (p3, t4), (p2, t2)\}$
$WarcTP' = \{(t1, p2, 1), (t2, p1, 1), (t3, p3, 1), (t4, p2, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p2, t3, 1), (p3, t4, 1), (p2, t2, 1)\}$
$M0' = \{(p1, 0), (p2, 1), (p3, 0)\}$
$M' = \{(p1, 0), (p2, 1), (p3, 0)\}$
$enabled' = \{\}$
$usePN' = build$
$event' = \{a, \overline{a}, b, \overline{b}\}$
$window' = \{(t2, (0, 4)), (t4, (5, 10))\}$
$condition' = \{(t1, a), (t2, \overline{a}), (t3, b), (t4, \overline{b})\}$
$pndate' = 0$

Après réalisation de l'opération *StartPlay*, à la date $date = 25$, le nouvel état est :

PNTE'

$P' = \{p1, p2, p3\}$
$T' = \{t1, t2, t3, t4\}$
$arcTP' = \{(t1, p2), (t2, p1), (t3, p3), (t4, p2)\}$
$arcPT' = \{(p1, t1), (p2, t3), (p3, t4), (p2, t2)\}$
$WarcTP' = \{(t1, p2, 1), (t2, p1, 1), (t3, p3, 1), (t4, p2, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p2, t3, 1), (p3, t4, 1), (p2, t2, 1)\}$
$M0' = \{(p1, 0), (p2, 1), (p3, 0)\}$
$M' = \{(p1, 0), (p2, 1), (p3, 0)\}$
$\mathbf{enabled'} = \{\mathbf{t2}, \mathbf{t3}\}$
$\mathbf{usePN'} = \mathbf{play}$
$event' = \{a, \overline{a}, b, \overline{b}\}$
$window' = \{(t2, (0, 4)), (t4, (5, 10))\}$
$condition' = \{(t1, a), (t2, \overline{a}), (t3, b), (t4, \overline{b})\}$
$\mathbf{pndate'} = \mathbf{25}$

L'opération *Play* est réalisée à la date $date = 36$ avec l'entrée $Event? = b$. La transition *Fired* est la transition $t3$ ($t3 \in enabled$ et $condition(t3) = b$ et $t3 \notin \mathrm{dom}(window)$). De plus :

$$arcPT^{\sim}(\!|\ \{t3\}\ |\!) \cap arcTP(\!|\ \{t3\}\ |\!) = \emptyset$$

$$arcPT^{\sim}(\!|\ \{t3\}\ |\!) \setminus arcTP(\!|\ \{t3\}\ |\!) = \{p2\}$$

$$arcTP(\!|\ \{t3\}\ |\!) \setminus arcPT^{\sim}(\!|\ \{t3\}\ |\!) = \{p3\}$$

$$arcPT^{\sim}(\!|\ \{t3\}\ |\!) \cup arcTP(\!|\ \{t3\}\ |\!) = \{p2, p3\}$$

d'où :

$$M'(p1) = M(p1) = 0$$

$$M'(p2) = M(p2) - WarcPT(p2 \mapsto t3) = 1 - 1 = 0$$

$$M'(p3) = M(p3) + WarcTP(t3 \mapsto p3) = 0 + 1 = 1$$

Le nouvel état est donc :

PNTE'

$P' = \{p1, p2, p3\}$
$T' = \{t1, t2, t3, t4\}$
$arcTP' = \{(t1, p2), (t2, p1), (t3, p3), (t4, p2)\}$
$arcPT' = \{(p1, t1), (p2, t3), (p3, t4), (p2, t2)\}$
$WarcTP' = \{(t1, p2, 1), (t2, p1, 1), (t3, p3, 1), (t4, p2, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p2, t3, 1), (p3, t4, 1), (p2, t2, 1)\}$
$M0' = \{(p1, 0), (p2, 1), (p3, 0)\}$
$\mathbf{M'} = \{(\mathbf{p1}, \mathbf{0}), (\mathbf{p2}, \mathbf{0}), (\mathbf{p3}, \mathbf{1})\}$
$\mathbf{enabled'} = \{\mathbf{t4}\}$
$usePN' = play$
$event' = \{a, \overline{a}, b, \overline{b}\}$
$window' = \{(t2, (0, 4)), (t4, (5, 10))\}$
$condition' = \{(t1, a), (t2, \overline{a}), (t3, b), (t4, \overline{b})\}$
$\mathbf{pndate'} = \mathbf{36}$

A la date $date = 42$ l'opération *Play* est à nouveau réalisée avec l'entrée $Event? = \overline{b}$. La transition *Fired* est la transition $t4$ ($t4 \in enabled$, $condition(t4) = \overline{b}$, $t4 \in \mathrm{dom}(window)$, $5 < (42 - 36)$ et $10 > (42 - 36)$). De plus :

$$arcPT^{\sim}(\!|\ \{t4\}\ |\!) \cap arcTP(\!|\ \{t4\}\ |\!) = \emptyset$$

$$arcPT^{\sim}(\!|\ \{t4\}\ |\!) \setminus arcTP(\!|\ \{t4\}\ |\!) = \{p3\}$$

$$arcTP(\!|\ \{t4\}\ |\!) \setminus arcPT^{\sim}(\!|\ \{t4\}\ |\!) = \{p2\}$$

$$arcPT^{\sim}(\!|\ \{t4\}\ |\!) \cup arcTP(\!|\ \{t4\}\ |\!) = \{p2, p3\}$$

d'où :

$$M'(p1) = M(p1) = 0$$

$$M'(p3) = M(p3) - WarcPT(p3 \mapsto t4) = 1 - 1 = 0$$

$$M'(p2) = M(p2) + WarcTP(t4 \mapsto p2) = 0 + 1 = 1$$

Le nouvel état est donc :

PNTE'

$P' = \{p1, p2, p3\}$
$T' = \{t1, t2, t3, t4\}$
$arcTP' = \{(t1, p2), (t2, p1), (t3, p3), (t4, p2)\}$
$arcPT' = \{(p1, t1), (p2, t3), (p3, t4), (p2, t2)\}$
$WarcTP' = \{(t1, p2, 1), (t2, p1, 1), (t3, p3, 1), (t4, p2, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p2, t3, 1), (p3, t4, 1), (p2, t2, 1)\}$
$M0' = \{(p1, 0), (p2, 1), (p3, 0)\}$
$\mathbf{M'} = \{(\mathbf{p1}, \mathbf{0}), (\mathbf{p2}, \mathbf{1}), (\mathbf{p3}, \mathbf{0})\}$
$\mathbf{enabled'} = \{\mathbf{t2}, \mathbf{t3}\}$
$usePN' = play$
$event' = \{a, \overline{a}, b, \overline{b}\}$
$window' = \{(t2, (0, 4)), (t4, (5, 10))\}$
$condition' = \{(t1, a), (t2, \overline{a}), (t3, b), (t4, \overline{b})\}$
$\mathbf{pndate'} = \mathbf{42}$

et ainsi de suite ...

5.2.5 Les modèles continus

La partie continue du système est décrite par des systèmes d'équations différentielles. Ces équations mathématiques ont une syntaxe et une sémantique parfaitement définies. Méta-modéliser ces équations n'apporterait pas plus de rigueur à ces équations. Il en est de même pour toutes les méthodes formelles issues du domaine continu. Par contre, il est nécessaire d'intégrer la partie continue au méta-modèle de la partie discrète pour obtenir un méta-modèle complet. Cette intégration se fera par l'intermésiaire d'un modèle générique permettant d'intégrer la sémantique de la partie continue au méta-modèle de l'intégration des deux aspects.

Le modèle générique

Quelle que soit l'équation différentielle utilisée dans notre système, elle a toujours la même forme : la dérivée par rapport au temps de la variable M_r est égale à une valeur constante q. Pour résoudre le système, il faut donner une valeur initiale M_{r0} et une valeur finale M_{r1}. Finalement, ce qui caractérise chaque système d'équation, c'est q et M_{r1}. La valeur de M_{r0} est donnée au moment de la résolution. Le schéma *Continue* représente l'état de ce système.

Continue

$q : \mathbb{R}$
$M_{r1} : \mathbb{R}$

Resolution

$\Xi Continue$
$M_{r0}? : \mathbb{R}$
$Tsol! : TIME$

$Tsol! = (M_{r1} - M_{r0}?)/q$

L'opération d'initialisation affecte, par exemple, la valeur 0 à q et à M_{r0}, l'opération de construction est très simple :

Construction

$\Delta Continue$
$valeurq? : \mathbb{R}$
$valeurM_r? : \mathbb{R}$

$q' = valeur?$

Un exemple

Soit à résoudre l'équation $\frac{dM_r(t)}{dt} = 2$ avec $M_{r0} = 4$ et $M_{r1} = 20$. L'état initial du système est :

Continue'

$q' = 0$
$M'_{r1} = 0$

L'opération *Construction* est réalisée avec $valeurq? = 2$ et $valeurM_r? = 20$. L'état devient :

Continue'

$q' = 2$
$M'_{r1} = 20$

L'opération *Resolution* donne comme résultat $Tsol! = 8$.

5.2.6 L'intégration

Le système final comporte un réseau PNTE pour la commande, un pour le système de référence discret et un ensemble d'équations pour le système de référence continu. De plus, une équation différentielle est liée à une place (fonction partielle *winpla*), et chaque équation permet de calculer la fenêtre temporelle liée à une transition : les transitions et les équations sont donc en relation (fonction *wintra*). Le schéma du système est :

System

$$
\begin{array}{l}
Commande : PNTE \\
Ref\_discret : PNTE \\
Ref\_cont : \mathbb{F}\ Continue \\
winpla : PLACE \pfun Continue \\
wintra : TRANSITION \pfun Continue \\
\hline
\mathrm{dom}\ winpla \subseteq Ref\_discret.P \\
\mathrm{dom}\ wintra \subseteq Ref\_discret.T
\end{array}
$$

Comme pour le schéma *PNTE*, et pour raccourcir les schémas d'opérations, nous introduisons les schémas $\Phi Commande$ et $\Phi Ref\_discret$ tels que :

$$\Phi Commande \mathrel{\widehat=} \Xi Commande \setminus (M, M', enabled, enabled', usePN, usePN', pndate, pndate')$$

$$\Phi Ref\_discret \mathrel{\widehat=} \Xi Ref\_discret \setminus (M, M', enabled, enabled', usePN, usePN', pndate, pndate')$$

Il n'est pas ici nécessaire de déclarer $\Delta Comande$ et $\Delta Ref\_discret$ car les opérations contiennent la déclaration de $\Delta System$ qui inclut ces deux déclarations.

L'opération *System_StartPlay* détermine, pour chaque PNTE, l'ensemble des transitions validées, initialise la date *pndate* et modifie la variable *usePN* pour que les réseaux puissent être joués.

System_StartPlay

$$
\begin{array}{l}
\Delta System \\
\Phi Commande \\
\Phi Ref\_discret \\
\Xi Time \\
\hline
\forall\, pnte : PNTE \bullet \\
\quad (pnte.enabled' = \{t : TRANSITION \mid (\forall\, p : PLACE \bullet \\
\qquad (p, t) \in pnte.arcPT' \land pnte.WacPT'(p, t) < pnte.M'(p)) \\
\qquad \lor\ t \notin \mathrm{ran}(pnte.arcPT') \bullet t\} \land \\
\quad pnte.usePN' = play \land \\
\quad pnte.pndate' = date)
\end{array}
$$

System_Play

$\Delta System$
$\Phi Commande$
$\Phi Ref\_discret$
$\Xi Time$
$Event? : EVENT$
$M_{r0}? : \mathbb{R}$
$Fired1, Fired2 : TRANSITION$

$\exists\, Fired1, Fired2 : TRANSITION \mid$
$Fired1 \in Commande.enabled \land$
$Commande.condition(Fired1) = Event? \land$
$((Fired1 \in \mathrm{dom}\, Commande.window \land$
$(\forall\, t1, t2 : TIME \mid (t1, t2) = Commande.window(Fired1) \bullet$
$t1 < (date - pndate) \land$
$t2 > (date - pndate))) \lor$
$Fired1 \notin \mathrm{dom}\, Commande.window)$
$\land$
$Fired2 \in Ref\_discret.enabled \land$
$Ref\_discret.condition(Fired2) = Event? \land$
$((Fired2 \in \mathrm{dom}\, Ref\_discret.window \land$
$(\forall\, t1, t2 : TIME \mid (t1, t2) = Ref\_discret.window(Fired2) \bullet$
$t1 < (date - pndate) \land$
$t2 > (date - pndate))) \lor$
$Fired2 \notin \mathrm{dom}\, Ref\_discret.window) \bullet$
$(\forall\, pnte : PNTE \bullet$
$Event? \in pnte.event \land$
$pnte.usePN = play \bullet$
$(\forall\, p : PLACE, t : TRANSITION \mid p \in pnte.P \land t \in \{Fired1, Fired2\} \bullet$
$(p \in pnte.arcPT^{\sim}(\!|\, \{t\} \,|\!) \cap pnte.arcTP(\!|\, \{t\} \,|\!)$
$\land\ pnte.M'(p) = pnte.M(p) - pnte.WarcPT(p \mapsto t) + pnte.WarcTP(t \mapsto p))$
$\lor$
$(p \in pnte.arcPT^{\sim}(\!|\, \{t\} \,|\!) \setminus pnte.arcTP(\!|\, \{t\} \,|\!)$
$\land\ pnte.M'(p) = pnte.M(p) - pnte.WarcPT(p \mapsto t))$
$\lor$
$(p \in pnte.arcTP(\!|\, \{t\} \,|\!) \setminus pnte.arcPT^{\sim}(\!|\, \{t\} \,|\!)$
$\land\ pnte.M'(p) = pnte.M(p) + pnte.WarcTP(t \mapsto p))$
$\lor$
$(p \notin pnte.arcPT^{\sim}(\!|\, \{t\} \,|\!) \cup pnte.arcTP(\!|\, \{t\} \,|\!)$
$\land\ pnte.M'(p) = pnte.M(p))) \land$
$pnte.enabled' = \{t : TRANSITION \mid (\forall\, p : PLACE \bullet$
$(p, t) \in pnte.arcPT' \land pnte.WacPT'(p, t) < pnte.M'(p))$
$\lor\ t \notin \mathrm{ran}(pnte.arcPT') \bullet t\} \land$
$pnte.usePN' = play \land$
$pnte.pndate' = date)$
$(\exists_1\, p : PLACE \mid Ref\_discret.M'(p) \neq 0 \land p \in \mathrm{dom}\, winpla \bullet$
$Ref\_discret.window' = \{wintra^{\sim}(winpla(p)) \mapsto$
$(0.95 * (winpla(p).M_{r1} - M_{r0}?)/winpla(p).q,$
$1.05 * (winpla(p).M_{r1} - M_{r0}?)/winpla(p).q)\} \lor$
$Ref\_discret.window' = \{\})$
$Commande.window' = \{\}$

5.3 Validation de ce méta-modèle

5.3.1 Instanciation du méta-modèle

L'exemple d'instanciation de notre méta-modèle est bien sûr l'exemple présenté dans la section 5.1.3. Nous avons donné des valeurs numériques aux grandeurs physiques, afin de pouvoir jouer le modèle. La figure suivante représente les trois parties *Commande*, *Ref_discret* et *Ref_cont*.

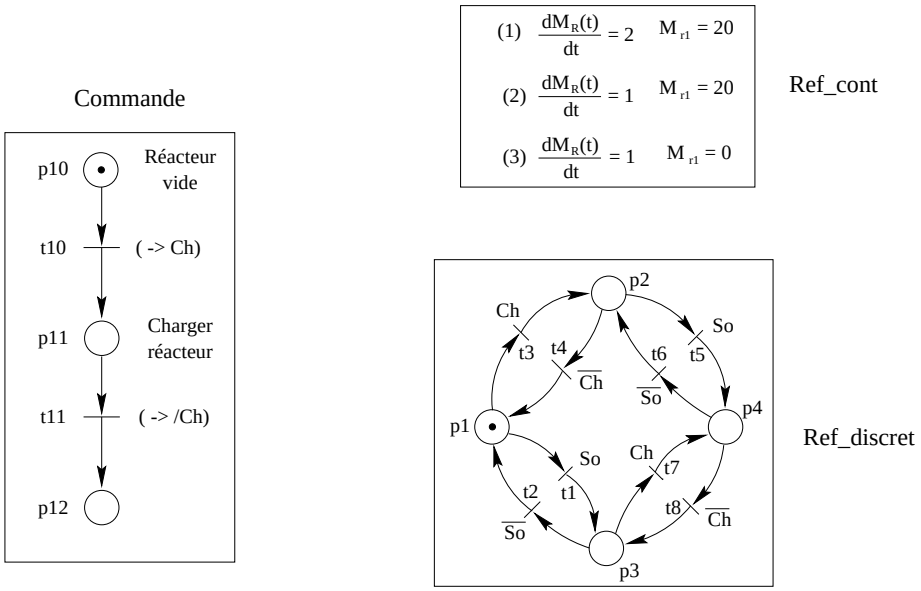


FIG. 5.5: *Exemple de validation : une unité de stockage*

Dans la suite, les instances sont présentées sous forme de schéma pour faciliter la lecture. Les schémas présentés maintenant correspondent à l'état après construction et avant le début des activités de jeu.

Commande'

$P' = \{p10, p11, p12\}$
$T' = \{t10, t11\}$
$arcTP' = \{(t10, p11), (t11, p12)\}$
$arcPT' = \{(p10, t11), (p11, t11)\}$
$WarcTP' = \{(t10, p11, 1), (t11, p12, 1)\}$
$WarcPT' = \{(p10, t11, 1), (p11, t11, 1)\}$
$M0' = \{(p10, 1), (p11, 0), (p12, 0)\}$
$M' = \{(p10, 1), (p11, 0), (p12, 0)\}$
$enabled' = \{\}$
$usePN' = build$
$event' = \{Ch, \overline{Ch}\}$
$window' = \{\}$
$condition' = \{(t10, Ch), (t11, \overline{Ch}\}$
$pndate' = 0$

Ref_discret'

$P' = \{p1, p2, p3, p4\}$
$T' = \{t1, t2, t3, t4, t5, t6, t7, t8\}$
$arcTP' = \{(t1, p3), (t2, p1), (t3, p2), (t4, p1), (t5, p4), (t6, p2), (t7, p4), (t8, p3)\}$
$arcPT' = \{(p1, t1), (p1, t3), (p2, t4), (p2, t5), (p3, t2), (p3, t7), (p4, t6), (p4, t8)\}$
$WarcTP' = \{(t1, p3, 1), (t2, p1, 1), (t3, p2, 1), (t4, p1, 1),$
$(t5, p4, 1), (t6, p2, 1), (t7, p4, 1), (t8, p3, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p1, t3, 1), (p2, t4, 1), (p2, t5, 1),$
$(p3, t2, 1), (p3, t7, 1), (p4, t6, 1), (p4, t8, 1)\}$
$M0' = \{(p1, 1), (p2, 0), (p3, 0), (p4, 0)\}$
$M' = \{(p1, 1), (p2, 0), (p3, 0), (p4, 0)\}$
$enabled' = \{\}$
$usePN' = build$
$event' = \{Ch, \overline{Ch}, So, \overline{So}\}$
$window' = \{\}$
$condition' = \{(t1, So), (t2, \overline{So}), (t3, Ch), (t4, \overline{Ch}), (t5, So), (t6, \overline{So}), (t7, Ch), (t8, \overline{Ch})\}$
$pndate' = 0$

Continue1'

$q' = 2$
$M'_{r1} = 20$

Continue2'

$q' = 1$
$M'_{r1} = 20$

Continue3'

$q' = 1$
$M'_{r1} = 0$

System'

$Commande' : PNTE$
$Ref\_discret' : PNTE$
$Ref\_cont' = \{Continue1, Continue2, Continue3\}$
$winpla' = \{(p2, Continue1), (p3, Continue3), (p4, Continue2)\}$
$wintra' = \{(t4, Continue1), (t2, Continue3), (t8, Continue2)\}$

Dans toutes les opérations qui suivent, les schémas *Continue1*, *Continue2*, *Continue3* et *System* ne sont pas modifiés : nous ne les réécrirons donc pas.

Après réalisation de l'opération *System_StartPlay*, à la date $date = 25$, le nouvel état est :

Commande'

$P' = \{p10, p11, p12\}$
$T' = \{t10, t11\}$
$arcTP' = \{(t10, p11), (t11, p12)\}$
$arcPT' = \{(p10, t11), (p11, t11)\}$
$WarcTP' = \{(t10, p11, 1), (t11, p12, 1)\}$
$WarcPT' = \{(p10, t11, 1), (p11, t11, 1)\}$
$M0' = \{(p10, 1), (p11, 0), (p12, 0)\}$
$M' = \{(p10, 1), (p11, 0), (p12, 0)\}$
$\mathbf{enabled'} = \{\mathbf{t10}\}$
$\mathbf{usePN'} = \mathbf{play}$
$event' = \{Ch, \overline{Ch}\}$
$window' = \{\}$
$condition' = \{(t10, Ch), (t11, \overline{Ch}\}$
$\mathbf{pndate'} = \mathbf{25}$

Ref_discret'

$P' = \{p1, p2, p3, p4\}$
$T' = \{t1, t2, t3, t4, t5, t6, t7, t8\}$
$arcTP' = \{(t1, p3), (t2, p1), (t3, p2), (t4, p1), (t5, p4), (t6, p2), (t7, p4), (t8, p3)\}$
$arcPT' = \{(p1, t1), (p1, t3), (p2, t4), (p2, t5), (p3, t2), (p3, t7), (p4, t6), (p4, t8)\}$
$WarcTP' = \{(t1, p3, 1), (t2, p1, 1), (t3, p2, 1), (t4, p1, 1),$
$(t5, p4, 1), (t6, p2, 1), (t7, p4, 1), (t8, p3, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p1, t3, 1), (p2, t4, 1), (p2, t5, 1),$
$(p3, t2, 1), (p3, t7, 1), (p4, t6, 1), (p4, t8, 1)\}$
$M0' = \{(p1, 1), (p2, 0), (p3, 0), (p4, 0)\}$
$M' = \{(p1, 1), (p2, 0), (p3, 0), (p4, 0)\}$
$\mathbf{enabled'} = \{\mathbf{t1}, \mathbf{t3}\}$
$\mathbf{usePN'} = \mathbf{play}$
$event' = \{Ch, \overline{Ch}, So, \overline{So}\}$
$window' = \{\}$
$condition' = \{(t1, So), (t2, \overline{So}), (t3, Ch), (t4, \overline{Ch}), (t5, So), (t6, \overline{So}), (t7, Ch), (t8, \overline{Ch})\}$
$\mathbf{pndate'} = \mathbf{25}$

L'opération *System_Play* est réalisée à la date $date = 36$ avec les entrées $Event? = Ch$ et $M_{r0}? = 4$. La transition *Fired1* est la transition $t10$ ($t10 \in Commande.enabled$, $Commande.condition(t10) = Ch$ et $t10 \notin Commande.window$), et la transition *Fired2* est la transition $t3$ ($t3 \in Ref\_discret.enabled$, $Ref\_discret.condition(t3) = Ch$ et $t3 \notin Ref\_discret.window$).

Commande'

$P' = \{p10, p11, p12\}$
$T' = \{t10, t11\}$
$arcTP' = \{(t10, p11), (t11, p12)\}$
$arcPT' = \{(p10, t11), (p11, t11)\}$
$WarcTP' = \{(t10, p11, 1), (t11, p12, 1)\}$
$WarcPT' = \{(p10, t11, 1), (p11, t11, 1)\}$
$M0' = \{(p10, 1), (p11, 0), (p12, 0)\}$
$\mathbf{M'} = \{(\mathbf{p10}, \mathbf{0}), (\mathbf{p11}, \mathbf{1}), (\mathbf{p12}, \mathbf{0})\}$
$\mathbf{enabled'} = \{\mathbf{t11}\}$
$usePN' = play\_$
$event' = \{Ch, \overline{Ch}\}$
$window' = \{\}$
$condition' = \{(t10, Ch), (t11, \overline{Ch}\}$
$\mathbf{pndate'} = \mathbf{36}$

Ref_discret'

$P' = \{p1, p2, p3, p4\}$
$T' = \{t1, t2, t3, t4, t5, t6, t7, t8\}$
$arcTP' = \{(t1, p3), (t2, p1), (t3, p2), (t4, p1), (t5, p4), (t6, p2), (t7, p4), (t8, p3)\}$
$arcPT' = \{(p1, t1), (p1, t3), (p2, t4), (p2, t5), (p3, t2), (p3, t7), (p4, t6), (p4, t8)\}$
$WarcTP' = \{(t1, p3, 1), (t2, p1, 1), (t3, p2, 1), (t4, p1, 1),$
$(t5, p4, 1), (t6, p2, 1), (t7, p4, 1), (t8, p3, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p1, t3, 1), (p2, t4, 1), (p2, t5, 1),$
$(p3, t2, 1), (p3, t7, 1), (p4, t6, 1), (p4, t8, 1)\}$
$M0' = \{(p1, 1), (p2, 0), (p3, 0), (p4, 0)\}$
$\mathbf{M'} = \{(\mathbf{p1}, \mathbf{0}), (\mathbf{p2}, \mathbf{1}), (\mathbf{p3}, \mathbf{0}), (\mathbf{p4}, \mathbf{0})\}$
$\mathbf{enabled'} = \{\mathbf{t4}, \mathbf{t5}\}$
$usePN' = play$
$event' = \{Ch, \overline{Ch}, So, \overline{So}\}$
$\mathbf{window'} = \{(\mathbf{t4}, (\mathbf{7.6}, \mathbf{8.4}))\}$
$condition' = \{(t1, So), (t2, \overline{So}), (t3, Ch), (t4, \overline{Ch}), (t5, So), (t6, \overline{So}), (t7, Ch), (t8, \overline{Ch})\}$
$\mathbf{pndate'} = \mathbf{36}$

A la date $date = 44.2$, l'opération *System_Play* est à nouveau réalisée, avec les entrées $Event? = \overline{Ch}$ et $M_{r0}? = 20$. La transition *Fired1* est la transition $t11$ ($t11 \in Commande.enabled$, $Commande.condition(t11) = \overline{Ch}$ et $t11 \notin Commande.window$), et la transition *Fired2* est la transition $t4$ ($t4 \in Ref\_discret.enabled$, $Ref\_discret.condition(t4) = \overline{Ch}$, $t4 \in Ref\_discret.window$, $7.6 < (44.2 - 36)$ et $8.4 > (44.2 - 36)$).

Commande'

$P' = \{p10, p11, p12\}$
$T' = \{t10, t11\}$
$arcTP' = \{(t10, p11), (t11, p12)\}$
$arcPT' = \{(p10, t11), (p11, t11)\}$
$WarcTP' = \{(t10, p11, 1), (t11, p12, 1)\}$
$WarcPT' = \{(p10, t11, 1), (p11, t11, 1)\}$
$M0' = \{(p10, 1), (p11, 0), (p12, 0)\}$
$\mathbf{M'} = \{(\mathbf{p10}, \mathbf{0}), (\mathbf{p11}, \mathbf{0}), (\mathbf{p12}, \mathbf{1})\}$
$\mathbf{enabled'} = \{\}$
$usePN' = play$
$event' = \{Ch, \overline{Ch}\}$
$window' = \{\}$
$condition' = \{(t10, Ch), (t11, \overline{Ch}\}$
$\mathbf{pndate'} = \mathbf{44.2}$

Ref_discret'

$P' = \{p1, p2, p3, p4\}$
$T' = \{t1, t2, t3, t4, t5, t6, t7, t8\}$
$arcTP' = \{(t1, p3), (t2, p1), (t3, p2), (t4, p1), (t5, p4), (t6, p2), (t7, p4), (t8, p3)\}$
$arcPT' = \{(p1, t1), (p1, t3), (p2, t4), (p2, t5), (p3, t2), (p3, t7), (p4, t6), (p4, t8)\}$
$WarcTP' = \{(t1, p3, 1), (t2, p1, 1), (t3, p2, 1), (t4, p1, 1),$
$(t5, p4, 1), (t6, p2, 1), (t7, p4, 1), (t8, p3, 1)\}$
$WarcPT' = \{(p1, t1, 1), (p1, t3, 1), (p2, t4, 1), (p2, t5, 1),$
$(p3, t2, 1), (p3, t7, 1), (p4, t6, 1), (p4, t8, 1)\}$
$M0' = \{(p1, 1), (p2, 0), (p3, 0), (p4, 0)\}$
$\mathbf{M'} = \{(\mathbf{p1}, \mathbf{1}), (\mathbf{p2}, \mathbf{0}), (\mathbf{p3}, \mathbf{0}), (\mathbf{p4}, \mathbf{0})\}$
$\mathbf{enabled'} = \{\mathbf{t1}, \mathbf{t3}\}$
$usePN' = play$
$event' = \{Ch, \overline{Ch}, So, \overline{So}\}$
$\mathbf{window'} = \{\}$
$condition' = \{(t1, So), (t2, \overline{So}), (t3, Ch), (t4, \overline{Ch}), (t5, So), (t6, \overline{So}), (t7, Ch), (t8, \overline{Ch})\}$
$\mathbf{pndate'} = \mathbf{44.2}$

et ainsi de suite ...

5.4 Conclusion

Dans ce chapitre, nous avons abordé deux sortes d'intégration entre deux méta-modèles. La première concerne l'intégration entre un langage et un langage dérivé : le réseau de Petri et le réseau de Petri temporel à événements. La seconde correspond à l'intégration de deux langages pour la construction d'un seul modèle à comportement dynamique.

La construction du méta-modèle du RdP temporel à événements à partir du méta-modèle du RdP a montré qu'une étude orientée données est insuffisante pour montrer les liens entre langages. En effet, cette étude n'aurait présenté le RdP temporel à événements que comme un RdP auquel ont été ajoutés des fenêtres temporelles et des événements (ce qui correspond aux schémas *PN* et *PNTE*). Notre étude a montré que l'ajout de ces signes a modifié la dynamique des modèles produits : le schéma *PNTE_Play* n'est pas déduit du schéma *Play* par simple ajout de prédicats sur les nouveaux signes. La dynamique a globalement été modifiée et donc le rapport entre les deux familles de réseaux est moins évident.

L'intégration au sein d'un même méta-modèle de deux langages temporels a nécessité une définition commune du temps et la spécification de liens entre signes des deux langages. Cet exemple a montré tout l'intérêt de l'utilisation des opérateurs sur schéma et du schéma comme type qui ont permis une spécification très claire. Il a encore démontré que l'aspect temporel des modèles pose plus de problèmes que l'aspect données, ce qui prouve bien que la méta-modélisation doit prendre en compte le comportement temporel des langages.

Dans la méthode que nous avons méta-modélisée, l'aspect continu repose sur des équations différentielles. Ces équations mathématiques ont bien évidemment une syntaxe et une sémantique sans ambiguïté. Elles ne nécessitent donc pas un méta-modèle qui n'apporterait rien. Il en est de même pour toutes les méthodes formelles issues du continu. Par contre, les méthodes hybrides, comme celle traitée ici, nécessitent d'intégrer la partie continue de la méthode au méta-modèle de la partie discrète. Dans ce cas, un modèle générique de la partie continue peut être intégré au méta-modèle de la partie discrète.

Conclusion générale

L'objectif de nos travaux est d'améliorer la qualité des systèmes automatisés de production en améliorant la qualité des activités de conception de ces systèmes. Ces activités produisent des modèles qui participent à la conception du SAP et qui servent également de supports de communication entre les intervenants des différentes activités. Pour concevoir un système de qualité, il faut avant tout produire des modèles de qualité. Pour obtenir cette qualité, notre approche a été de proposer la méta-modélisation formelle des activités de modélisations et notamment des modèles produits à chacune des étapes de conception des SAP.

Le premier résultat important de nos travaux est d'apporter un ensemble cohérent de définitions relatives à l'activité de modélisation d'un SAP : symbole, langage, modèle, méta-modèle, méthode, concept, théorie, syntaxe abstraite, syntaxe concrète, sémantique interne, sémantique externe. Ces définitions permettent de préciser quels sont les éléments manipulés durant l'activité de modélisation et de quelles façons ils le sont. Nous avons ainsi défini les éléments utilisés dans chaque activité de modélisation, le langage et la méthode, ainsi que le résultat de cette activité: le modèle. Nous avons aussi clarifié les quatre aspects de l'étude d'un langage: la syntaxe abstraite, la syntaxe concrète, la sémantique interne et la sémantique externe. Enfin, nous avons classifié les différentes opérations qui constituent les méthodes : construction, vérification, importation, exportation, validation et jeu. Cet apport est d'autant plus important que nous pensons qu'il permet à lui seul de rendre plus rigoureuse l'activité de modélisation en apportant aux concepteurs de SAP un cadre sémantique précis. De plus, il peut permettre de mieux situer les apports de futurs travaux de méta-modélisation.

Le second résultat de nos travaux est d'avoir montré la faisabilité et l'intérêt de la méta-modélisation formelle de l'activité de modélisation des SAP. L'utilisation du langage Z permet de spécifier, dans un même méta-modèle, aussi bien la structure des symboles des langages, que la construction, la vérification, l'importation, l'exportation, la validation et le jeu de modèles. Cette intégration de tous les aspects de l'activité de modélisation garantit la cohérence des méta-modèles. L'utilisation d'un langage formel comme méta-langage permet de valider et de vérifier les méta-modèles, assurant ainsi de leur rigueur et de leur

respect des besoins. Par l'intermédiaire de la méthode méta-modélisée dans le chapitre 5, nous avons, en plus, exploré une nouvelle voie de méta-modélisation : l'intégration de l'activité de jeu de deux langages au sein d'un même modèle. Cette méthode nous a en plus permis d'étendre notre approche, dédiée à l'origine aux systèmes à événements discrets, à l'étude des systèmes hybrides.

Aujourd'hui, de nombreux langages de modélisation sont utilisés pour la conception des SAP. Par contre, très peu de ces langages sont intégrés dans des méthodes permettant d'aider à la construction des modèles. A partir de nos travaux, trois voies peuvent être explorées.

La première consiste à détailler la succession des opérations élémentaires permises lors de la construction d'un modèle. A chaque étape de la construction d'un modèle, tous les symboles et les liens ne sont pas utilisables. En précisant ces restrictions, le méta-modèle permettrait de proposer un guide méthodologique d'utilisation d'un langage pour construire des modèles. De plus, le méta-modèle contrôlerait, à chaque opération, les propriétés du modèle, pouvant aller jusqu'à indiquer les modifications à effectuer pour les vérifier.

Dans notre étude de la syntaxe, nous nous sommes surtout intéressés à l'étude de la syntaxe abstraite. La syntaxe concrète n'est pas pour autant négligeable. Nous avons, par exemple, l'habitude de lire de gauche à droite et de haut en bas. Cette habitude fait que, dans un modèle d'actigramme SADT, nous considérons que les activités qui sont dans la plus grande diagonale (d'en haut à gauche à en bas à droite) sont les plus importantes du diagramme. Une deuxième voie d'amélioration de la construction de modèles est donc l'étude de la syntaxe concrète. A partir des méta-modèles tels que nous les avons définis, l'étude de la syntaxe concrète vient compléter l'étude des langages et des méthodes de construction.

Une troisième voie d'amélioration de la construction des modèles est l'utilisation de modèles génériques. Pour donner une sémantique externe aux langages, il faut définir des liens entre symboles de plusieurs langages ou définir des liens entre symboles d'un langage et concepts de la théorie associée au système modélisé. Lorsqu'elle est modélisée, cette théorie constitue un modèle générique du système. Lorsque le méta-modèle d'un langage et le modèle générique du système sont exprimés dans un même langage, ils peuvent être intégrés. Cette intégration permet de restreindre le langage à une utilisation ayant un sens vis-à-vis du système, et donc d'aider à la modélisation.

Annexe A

Présentation de Z

Le langage Z fut inventé par Jean-Raymond Abrial en France et développé par une équipe du Programming Research Group (PRG) de l'université d'Oxford, en Angleterre, dirigée par le professeur C.A.R. Hoare [Lightfoot94]. Ce langage est basé sur une théorie des ensembles, sur la logique du premier ordre et sur la notation par schémas qui permet de structurer les spécifications.

La première section de cette annexe présente un exemple de spécification écrite avec Z. Cet exemple est destiné à rappeler les bases du langage. Les sections suivantes reprennent chacun des aspects de Z pour en expliciter les symboles utilisés.

A.1 Présentation sur un exemple

L'exemple de modèle Z présenté ici est un extrait de la spécification de l'interface d'utilisation d'un système de gestion de fichier [Woodcock et al.96]. Cet exemple caractéristique de l'utilisation de Z nous permet de présenter succinctement les différents aspects d'un modèle Z.

$[clé, donnée]$

Fichier

$contenu : clé \pfun donnée$

InitialisationFichier

$Fichier'$

$contenu' = \emptyset$

$Lire_D$

$\Xi Fichier$
$k? : clé$
$d! : donnée$

$k? \in \mathrm{dom}\, contenu$
$d! = contenu(k?)$

$Ecrire_D$

$\Delta Fichier$
$k? : clé$
$d? : donnée$

$k? \in \mathrm{dom}\, contenu$
$contenu' = contenu \oplus \{k? \mapsto d?\}$

Fichier, *InitialisationFichier*, $Lire_D$ et $Ecrire_D$ sont des schémas. *Fichier* représente l'invariant de l'état du système modélisé. *InitialisationFichier* est l'opération d'initialisation de l'état, $Lire_D$ représente l'opération de lecture de l'état du système, $Ecrire_D$ représente l'écriture dans l'état du système.

$[clé, donnée]$ sont les types utilisés. Ils permettent de définir les individus et les ensembles (les relations et les fonctions sont des ensembles dans Z). $contenu : clé \pfun donnée$ est une fonction définie à partir de ces deux types. En Z les déclarations d'individus et d'ensembles sont locales (internes au schéma). C'est pourquoi les schémas décrivant une opération doivent faire référence au schéma décrivant l'état. Une opération décrit deux états : l'état avant (*Fichier*) et l'état après ($Fichier'$). L'opération d'initialisation (*InitialisationFichier*) ne décrit que l'état après. L'inclusion de $\Xi Fichier$ dans le schéma $Lire_D$ signifie que cette opération ne modifie pas l'état ($Fichier' = Fichier$). $\Delta Fichier$ signifie que l'opération $Ecrire_D$ modifie l'état ($Fichier' \neq Fichier$). Les éléments (individus et ensembles) dont le nom se termine par un point d'interrogation ($k?$ et $d?$) représentent les entrées de l'opération. Les éléments dont le nom se termine par un point d'exclamation ($d!$) sont les sorties de l'opération. L'état après l'opération des individus et des ensembles est décrit en ajoutant une apostrophe à leur nom.

Un schéma est composé de deux parties. La partie supérieure – au dessus de la barre horizontale (tout le schéma s'il n'y en a pas) – est la partie déclarative. Elle permet de déclarer (c'est à dire typer) les individus et ensembles. La partie inférieure est la partie prédicative. Elle permet – par l'utilisation d'opérations ensemblistes et de la logique du premier ordre – de définir la précondition et la postcondition de l'opération. La précondition correspond à la condition qui doit être respectée pour que l'opération ait lieu. La postcondition correspond à la description de l'état après l'opération.

Les deux opérations $Lire_D$ et $Ecrire_D$ ont la même précondition : $k?$ doit faire partie

du domaine de la fonction *contenu* ($k? \in \mathrm{dom}\, contenu$). L'opération $Lire_D$ ne modifie pas l'état : elle n'a qu'une entrée et une sortie. $d!$ est l'image de $k?$ par la fonction *contenu*. L'opération $Ecrire_D$ modifie l'état : elle comprend deux entrées et une description de l'état de la fonction *contenu* après opération ($contenu'$). La fonction *contenu*, après opération ($contenu'$), est égale à ce qu'elle était avant ($contenu$), plus la paire $(k?, d?)$($k?$ de type *clé*, $d?$ de type *donnée*).

Les deux prédicats ($k? \in \mathrm{dom}\, contenu$) et ($d! = contenu(k?)$) de l'opération $Lire_D$ sont écrits sur deux lignes. Cela revient à écrire ($k? \in \mathrm{dom}\, contenu \wedge d! = contenu(k?)$). Ce prédicat est donc vrai si les deux prédicats ($k? \in \mathrm{dom}\, contenu$) et ($d! = contenu(k?)$) sont vrais. Ce schéma n'exprime donc pas ce qui se passe si ($k? \in \mathrm{dom}\, contenu$) est faux. Il en est de même pour le schéma $Ecrire_D$.

Cet exemple peut être étendu pour tenir compte de tous les cas possibles d'une précondition. Par exemple l'opération d'écriture peut être spécifiée par :

$Rapport ::= clé\_utilisée \mid clé\_non\_utilisée \mid ok$

ErreurClé

$\Xi Fichier$
$k? : Clé$
$r! : Rapport$

CléNonUtilisée

ErreurClé

$k? \notin \mathrm{dom}\, contenu$
$r! = clé\_non\_utilisée$

Succès

$r! : Rapport$

$r! = ok$

$Ecrire \mathrel{\widehat{=}} (Ecrire_D \wedge Succès) \vee CléNonUtilisée$

Rapport est un nouveau type, un type libre. Cela signifie que tout individu de type *Rapport* (par exemple $r!$) ne pourra prendre que l'une des trois valeurs énumérées : *clé_utilisée*, *clé_non_utilisée* ou *ok*.

Le schéma *Ecrire* est égal aux schémas $Ecrire_D$ et *Succès* ou *CléNonUtilisée* définis précédemment. Cela revient à écrire :

Ecrire

$contenu, contenu' : Clé \pfun Donnée$
$k? : Clé$
$d? : donnée$
$r! : Rapport$

$(k? \in \mathrm{dom}\ contenu \land$
$\quad contenu' = contenu \oplus \{k? \mapsto d?\} \land$
$\quad r! = ok)$
$\lor$
$(k? \notin \mathrm{dom}\ contenu \land$
$\quad contenu' = contenu$
$\quad r! = clé\_non\_utilisée)$

La première écriture (définition du schéma *Ecrire* par utilisation d'opérateurs) permet d'étudier d'abord les résultats souhaités des opérations avant de compléter la spécification par l'étude de tous les cas possibles. Elle permet ainsi une construction modulaire tout en étant aussi lisible que la seconde (forme développée du schéma *Ecrire*).

Cet exemple rapide nous a permis de présenter différents aspects du langage Z. Cette présentation est loin d'être exhaustive. Les parties suivantes ont pour but d'aller plus loin dans la présentation des symboles utilisés en Z.

A.2 Logique des prédicats du premier ordre

La logique utilisée dans Z est une logique à deux valeurs : *vrai* et *faux*. Il n'existe pas, comme dans VDM par exemple, de valeur *indéterminé*. Si la valeur d'un prédicat ne peut pas être déterminée, ce prédicat peut être interprété comme *vrai* ou *faux*.

A.2.1 Logique des propositions

La logique des propositions est un sous-ensemble de la logique des prédicats. Cette logique comporte des opérateurs permettant de construire un prédicat à partir d'autres prédicats. Nous présentons ici ces opérateurs (dans cette partie, p, q et r sont des prédicats) :

$r = \neg\ p$	r est égal à la négation de p (r égal non p). La valeur de r est *vrai* (resp. *faux*) si p a comme valeur *faux* (resp. *vrai*)
$r = p \land q$	r est égal à p et q. r est vrai si p et q ont chacun la valeur *vrai*, sinon r a la valeur *faux*
$r = p \lor q$	r est égal à p ou q. r a la valeur *vrai* si au moins un des prédicats p et q a la valeur *vrai*

$r = p \Rightarrow q$ r est égal à p implique q. L'implication est définie de la même façon que $\neg\ p \vee q$. r a la valeur *vrai* si $p$et q ont la valeur *vrai* ou si p a la valeur *faux*

$r = p \Leftrightarrow q$ r est égal à p équivaut à q. L'équivalence est définie de la même façon que $p \Rightarrow q \wedge q \Rightarrow p$. r a la valeur *vrai* si p et q ont la même valeur

p	q	$\neg\ p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
vrai	vrai	faux	vrai	vrai	vrai	vrai
vrai	faux	faux	faux	vrai	faux	faux
faux	vrai	vrai	faux	vrai	vrai	faux
faux	faux	vrai	faux	faux	vrai	vrai

A.2.2 Quantification

La logique des prédicats étend la logique propositionnelle avec la quantification des variables :

$\forall X \bullet p$ quel que soit X, alors p

$\exists X \bullet p$ il existe X pour que p ait la valeur *vrai*

$\exists_1 X \bullet p$ il existe un unique X pour lequel p a la valeur *vrai*

A.3 Théorie des ensembles

Le langage Z n'est en fait pas basé directement sur une théorie des ensembles mais sur les axiomes de Zermelo-Fraenkel, sans l'axiome du choix et sans le schéma de remplacement [Monin96]. Ceci permet d'assurer l'existence d'une classe d'ensembles cloisonnés appelés types. Ces types sont disjoints. Il n'est pas possible de les réunir ou de former un ensemble comportant des individus choisis parmi plusieurs d'entre eux. Il existe cependant quelques opérateurs permettant de construire des types composés.

L'utilisation des types dans le langage Z est très restrictive. Par contre les types permettent de vérifier que les opérations sur les ensembles ne sont pas des non sens et ils aident donc à la structuration des spécifications.

A.3.1 Les types

Les types sont de quatre sortes : les types construits, les types libres, les types de base et les schémas.

Le seul type construit est $\mathbb{Z}$. $\mathbb{N}$ ou $\mathbb{N}_1$ ne sont pas des types (bien qu'ils soient régulièrement utilisés tels quels), mais des sous-ensembles de $\mathbb{Z}$. $\mathbb{R}$ n'est pas non plus un type, bien que certains auteurs proposent de l'inclure dans le langage Z en tant que type construit.

Les types de base permettent de déclarer des types sans se soucier de ce que sont réellement les individus (ou presque). Par exemple, si nous voulons utiliser des ensembles de villes et de personnes dans une spécification, nous devons déclarer les types :

$$[VILLE, PERSONNE]$$

Il faut cependant supposer que les individus sont définis de manière unique. Si nous identifions les personnes par leur nom, « dupont » ne peut représenter qu'un seul individu d'un ensemble de type PERSONNE. Il pourrait tout aussi bien représenter un individu d'un ensemble de type VILLE, par contre il ne peut pas exister en même temps un individu « durand » d'un ensemble de type PERSONNE et un autre d'un ensemble de type VILLE.

Les types libres sont des types dont les individus possibles des ensembles de ce type sont énumérés. Par exemple, si nous voulons que les ensembles de villes ne comportent que certaines villes, nous pouvons déclarer le type :

$$Ville ::= Lille \mid Lyon \mid Marseille \mid Paris \mid Toulouse$$

Il est également possible de construire un type libre à partir d'autres types ou de lui-même ([Spivey94], page 83, pour plus de détails) :

$$TypeLibre2 ::= élément1 \mid constr1\langle\langle TypeLibre1\rangle\rangle \mid constr2\langle\langle TypeLibre2\rangle\rangle \mid ...$$

Cette possibilité est très utile pour décrire des structures récursives telles que les arbres.

Un schéma peut également être utilissé comme type (voir section A.4.4). La déclaration d'un type schéma consiste à écrire le schéma, comme n'importe quel schéma (voir section A.4) pour tout ce qui concerne les schémas).

Un type peut également être déclaré par l'utilisation d'une définition abrégée. Elle permet de :

– redéfinir un type pour un besoin particulier :

$$DATE \mathrel{\widehat=} \mathbb{N}$$

– de définir un type en compréhension (voir section sur les ensembles) :

$$\mathbb{B} \mathrel{\widehat=} \{x : \mathbb{Z} \mid x = 0 \lor x = 1\}$$

– de définir un type comme le produit cartésien de plusieurs types :

$$PERSONNE \mathrel{\widehat{=}} NOM \times PRENOM$$

A.3.2 Les ensembles

A partir de maintenant, le terme ensemble ne comprendra pas les types présentés précédemment. Un ensemble est une collection d'individus. Tout ensemble ou tout individu doit être déclaré, c'est à dire typé. Par exemple :

$durand : PERSONNE$	signifie que *durand* est une variable qui prend comme valeur un individu de type *PERSONNE* (pour faire plus court, on écrira plutôt par la suite : *durand* est un individu de type *PERSONNE*),
$homme : \mathbb{P}\, PERSONNE$	signifie que *homme* est une variable qui prend comme valeur un ensemble de type *PERSONNE* (pour faire plus court, on écrira plutôt par la suite : *homme* est un ensemble de type *PERSONNE*).

Il existe d'autres déclarations possibles :

$homme : \mathbb{F}\, PERSONNE$	signifie que *homme* est un ensemble fini de type *PERSONNE*,
$homme : \mathbb{P}_1\, PERSONNE$	signifie que *homme* est un ensemble non vide de type *PERSONNE*,
$homme : \mathbb{F}_1\, PERSONNE$	signifie que *homme* est un ensemble fini non vide de type *PERSONNE*.

Un ensemble peut être déclaré par énumération : $A \mathrel{\widehat{=}} \{1, 2, 3, 5, 7\}$ signifie que l'ensemble A est égal à l'ensemble comprenant les individus 1, 2, 3, 5 et 7. L'ensemble vide peut s'écrire de deux manières : $\emptyset$ et $\{\}$. En fait, il n'existe pas un seul ensemble vide, mais un par type défini, cependant son écriture et sa définition sont génériques.

Les individus de type produit cartésien de plusieurs types $(X \times Y)$ sont appelés des n-uplets. Ils s'écrivent (x, y, z) (ou $x \mapsto y \mapsto z$, voir section relations), et un n-uplet est ordonné. On appelle couple un n-uplet formé de deux types, triplet un n-uplet formé de trois types, quadruplet un n-uplet formé de quatre types, etc.

Les opérateurs habituels d'opération sur les individus et sur les ensembles sont utilisables en Z. On retrouve ainsi :

$x = y$	l'expression x est égale à l'expression y
$x \neq y$	l'expression x n'est pas égale à l'expression y
$a \in A$	a est un individu de l'ensemble A
$a \notin A$	a n'est pas un individu de l'ensemble A
$A \subseteq B$	l'ensemble A est un sous-ensemble de l'ensemble B
$A \subset B$	A est un sous-ensemble propre de B $(A \subseteq B \land A \neq B)$
$C = A \cup B$	l'ensemble C est égal à l'union des ensembles A et B
$C = A \cap B$	l'ensemble C est égal à l'intersection des deux ensembles A et B
$C = A \setminus B$	l'ensemble C est égal à la différence des deux ensembles A et B $\forall A, B : \mathbb{P}\, X \bullet A \setminus B = \{x : X \mid x \in A \land x \notin B\}$
$B = \bigcup A$	l'ensemble B est égal à l'union généralisée de l'ensemble d'ensembles A $\forall A : \mathbb{P}(\mathbb{P}\, X) \bullet \bigcup A = \{x : X \mid (\exists S : A \bullet x \in S)\}$
$B = \bigcap A$	l'ensemble B est égal à l'intersection généralisée de l'ensemble d'ensembles A $\forall A : \mathbb{P}(\mathbb{P}\, X) \bullet \bigcap A = \{x : X \mid (\forall S : A \bullet x \in S)\}$
$n = \#A$	n est égal à la cardinalité de l'ensemble A (nombre d'individus de A)

A.3.3 Les relations

Dans le langage Z, une relation est un ensemble. Écrire $R : A \leftrightarrow B$ signifie que la relation R est un sous-ensemble du produit cartésien de A et B :

$$R : A \leftrightarrow B \mathrel{\widehat{=}} \mathbb{P}\, A \times B$$

Un individu défini à partir du symbole relation s'écrit plutôt en utilisant la correspondance binaire $a \mapsto b$, tandis qu'un individu défini à partir du produit cartésien s'écrit plutôt comme un couple (a, b). Cependant les deux écritures sont équivalentes :

$$R : a \mapsto b \mathrel{\widehat{=}} (a, b)$$

Une relation $R : A \leftrightarrow B$ implique un sous-ensemble des individus des types A et B. Le sous-ensemble de A impliqué dans la relation R est le domaine de la relation ($\mathrm{dom}\, R$). Le sous-ensemble de B impliqué dans la relation R est le co-domaine de la relation ($\mathrm{ran}\, R$). L'ensemble des individus reliés aux individus de l'ensemble A du domaine de R s'appelle l'image de A, elle s'écrit $R(\!|\ A\ |\!)$. L'image d'un ensemble $\{a, b\}$ s'écrit $R(\!|\ \{a, b\}\ |\!)$. Les opérateurs sur relations sont présentés ici :

$R : \mathrm{id}\, A$ la relation R est la relation d'identité sur A : elle relie chaque individu de type A à lui-même

$P = Q \fatsemi R$ la relation $P : X \leftrightarrow Z$ est la relation de composition des relations $Q : X \leftrightarrow Y$ et $R : Y \leftrightarrow Z$. Elle relie un individu x de X à un individu z de Z si et seulement si il existe au moins un individu y de Y qui est relié à x par Q et qui est aussi relié à z par R

$P = Q \circ R$ la relation $P : X \leftrightarrow Z$ est la relation de composition arrière des relations $Q : X \leftrightarrow Y$ et $R : Y \leftrightarrow Z$. On a l'égalité : $Q \circ R = R \fatsemi Q$

$P = A \lhd R$ la relation P est la relation R avec une restriction de son domaine à l'ensemble A. La relation P relie x à y si et seulement si R relie x à y et si x appartient à A

$P = R \rhd A$ la relation P est la relation R avec une restriction de son codomaine à l'ensemble A. La relation P relie x à y si et seulement si R relie x à y et si y appartient à A

$P = A \lhd\!\!\!\!- R$ la relation P est la relation R avec une antirestriction de son domaine à l'ensemble A. La relation P relie x à y si et seulement si R relie x à y et si x n'appartient pas à A

$P = R \rhd\!\!\!\!- A$ la relation P est la relation R avec une antirestriction de son codomaine à l'ensemble A. La relation P relie x à y si et seulement si R relie x à y et si y n'appartient pas à A

$P = R^{-1}$ la relation P est la relation inverse de R. La relation P relie x à y si et seulement si la relation R relie y à x

$P = Q \oplus R$ la relation P est l'écrasement de la relation Q par la relation R. La relation P relie tout individu du domaine de R aux mêmes individus que la relation R, et tout individu du domaine de Q et ne faisant pas partie du domaine de R aux mêmes individus que la relation Q

A.3.4 Les fonctions

Dans Z, les fonctions sont des cas particuliers de relations pour lesquelles chaque individu du domaine a au plus un individu image. Le cas le plus général de la fonction est la fonction partielle, représentée par $\pfun$. Des cas particuliers de fonctions partielles existent : fonction totale, injection partielle, injection totale, surjection partielle, surjection totale, bijection. La figure suivante présente les différentes fonctions dont les définitions sont ensuite présentées.

$f : A \pfun B$ fonction partielle f du type A vers le type B. Tout individu du domaine de f a au plus un individu image

$$A \pfun B \mathrel{\widehat=} \{f : A \rel B \mid (\forall\, a : A; b_1, b_2 : B \bullet (a \mapsto b_1) \in f \land (a \mapsto b_2) \in f \Rightarrow b_1 = b_2)\}$$

$f : A \fun B$ fonction totale f du type A vers le type B. Le domaine de la fonction est égal à l'ensemble de départ A. Tout individu de A a un et un seul individu image

$A \fun B \mathrel{\widehat=} \{f : A \pfun B \mid \mathrm{dom}\, f = A\}$

$f : A \pinj B$ injection partielle f du type A vers le type B. Une injection partielle est une fonction partielle pour laquelle deux individus distincts du domaine ne peuvent avoir le même individu image

$A \pinj B \mathrel{\widehat=} \{f : A \pfun B \mid (\forall\, a_1, a_2 : \mathrm{dom}\, f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2)\}$

$f : A \inj B$ injection totale f du type A vers le type B. Une injection totale est une fonction totale pour laquelle deux individus distincts de A ne peuvent avoir le même individu image

$A \inj B \mathrel{\widehat=} (A \pinj B) \cap (A \fun B)$

$f : A \psurj B$ surjection partielle f du type A vers le type B. Une surjection partielle est une fonction partielle pour laquelle l'ensemble image de f est B

$A \psurj B \mathrel{\widehat=} \{f : A \pfun B \mid \mathrm{ran}\, f = B\}$

$f : A \surj B$ surjection totale f du type A vers le type B. Une surjection totale est à la fois une fonction totale (tout individu de A a un et un seul individu image) et une surjection partielle (l'image de f est B)

$A \surj B \mathrel{\widehat=} (A \psurj B) \cap (A \fun B)$

$f : A \bij B$ bijection f du type A vers le type B. Une bijection est à la fois une surjection totale et une injection totale : a tout individu du domaine de f (A) correspond un et un seul individu de l'image de f (B)

$A \bij B \mathrel{\widehat=} (A \surj B) \cap (A \inj B)$

$f : A \ffun B$ fonction finie partielle f du type A vers le type B. f est une fonction partielle dont le domaine est égal à un sous-ensemble fini de A

$A \ffun B \mathrel{\widehat=} \{f : A \pfun B \mid \mathrm{dom}\, f \in \mathbb{F}\, A\}$

$f : A \finj B$ injection finie partielle f du type A vers le type B. f est à la fois une fonction finie partielle et une injection partielle

$A \finj B \mathrel{\widehat=} (A \ffun B) \cap (A \pinj B)$

A.3.5 Les suites

Pour modéliser les suites, les vecteurs, etc, il existe dans le langage Z une fonction prédéfinie : la suite (*sequence* en anglais, souvent traduit par séquence en français). Une suite s de type T est une fonction partielle de $\mathbb{N}$ vers T, et de domaine les individus 1, 2

... $\#s$ (cardinal s). La suite vide est notée: $\langle\rangle$. Il existe également la fonction suite non vide: seq_1, et de nombreux opérateurs sur les suites, présentés ci-dessous.

$s : \text{seq}\ T$	suite finie de type T $\text{seq}\ T \mathrel{\widehat=} \{s : \mathbb{N} \pfun T \mid \text{dom}\ s = 1 \mathinner{..} \#s\}$
$s : \text{seq}_1\ T$	suite finie non vide: $\text{seq}_1\ T \mathrel{\widehat=} \text{seq}\ T \setminus \{\langle\rangle\}$
$s \frown t$	la suite s est concaténée avec la suite t (toutes les deux de types T). Cela revient à créer une nouvelle suite dont les premiers individus sont les individus de s et les derniers individus les individus de t $\forall s, t : \text{seq}\ T \bullet s \frown t = s \cup \{n : \text{dom}\ t \bullet n + \#s \mapsto t(n)\}$
$t = rev\ s$	retournement de la suite s. La suite t est une suite contenant les mêmes individus que s mais dans l'ordre inverse.
$head\ s$	premier individu d'une suite non vide $\forall s : \text{seq}_1\ T \bullet head\ s = s(1)$
$last\ s$	dernier individu d'une suite non vide $\forall s : \text{seq}_1\ T \bullet last\ s = s(\#s)$
$t = tail\ s$	la suite t est égale à la queue de la suite s: elle contient tous les individus de s sauf le premier
$t = front\ s$	la suite t est égale à l'avant de la suite s: elle contient tous les individus de s sauf le dernier
$t = A \upharpoonleft s$	t est une suite égale à l'extraction de la suite s des individus d'indice les indices de l'ensemble A, avec le même ordre que dans s. Par exemple $\{2, 5\} \upharpoonleft \langle S, U, I, T, E\rangle = \langle U, E\rangle$
$t = s \upharpoonright A$	t est une suite égale au filtrage de la suite s par l'ensemble A : la suite t ne comprend que les individus compris dans s et A, dans le même ordre que dans s. Par exemple $\langle S, U, I, T, E\rangle \upharpoonright \{T, U, E, R\} = \langle U, T, E\rangle$
$s = squash\ f$	la suite s est égale au compactage de la fonction f (définie sur des entiers strictement positifs). La fonction f. Par exemple $squash\ \{2 \mapsto S, 5 \mapsto U, 7 \mapsto I, 8 \mapsto T, 9 \mapsto E\} = \{1 \mapsto S, 2 \mapsto U, 3 \mapsto I, 4 \mapsto T, 5 \mapsto E\} = \langle S, U, I, T, E\rangle$
s prefix t	la suite s est en tête de la suite t. Cela signifie qu'il existe une suite r telle que $s \frown r = t$. Par exemple $\langle S, U, I\rangle$ prefix $\langle S, U, I, T, E\rangle$ a pour valeur *vrai*

s suffix t	la suite s est en queue de la suite t. Cela signifie qu'il existe une suite r telle que $r \frown s = t$. Par exemple $\langle I, T, E\rangle$ suffix $\langle S, U, I, T, E\rangle$ a pour valeur *vrai*
s in t	la suite s est dans la suite t. Cela signifie qu'il existe deux suites r et u telles que $r \frown s \frown u = t$. Par exemple $\langle U, I, T\rangle$ in $\langle S, U, I, T, E\rangle$ a pour valeur *vrai*
$r = \frown/\langle s, t\rangle$	la suite r est la concaténation distribuée des suites s et t. Les premiers individus de r sont les individus de s, les derniers sont les individus de t. En fait $\frown/\langle s, t\rangle = s \frown t$
disjoint $\langle A, B\rangle$	disjonction des ensembles A et B : cette disjonction a comme valeur *vrai* si et seulement si les ensembles A et B ont une intersection nulle. disjoint $\langle A, B\rangle \Leftrightarrow A \cap B = \emptyset$
$\langle A, B\rangle$ partition C	la suite constituée des individus des ensembles A et B est une partition de l'ensemble C. Ceci n'est vrai que si l'intersection de A et B est vide et si l'union de A et B est l'ensemble C $\langle A, B\rangle$ partition $C \Leftrightarrow A \cap B = \emptyset \wedge A \cup B = C$

A.3.6 Les multi-ensembles ou sacs

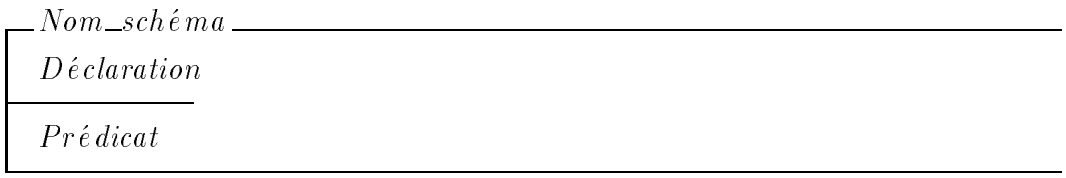
Les opérateurs suivants ne sont ici que présentés, le lecteur se reportera à [Spivey94] pour plus d'informations.

bag	multi-ensemble ou sac
count	comptage
$[\![]\!]$	sac vide
$[\![a_1, ..., a_n]\!]$	écriture par extension d'un sac
in	appartenance de sac
$\sqsubseteq$	relation de sous-sacs
$\uplus$	union de sacs
items s	sac des individus d'une suite

A.4 Les structures

A.4.1 Les déclarations

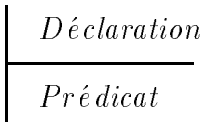
La structure de base d'une spécification en langage Z est le schéma ordinaire, appelé communément schéma. Sa forme verticale est présentée ci-après :



Un schéma est identifié par son nom (*Nom_schéma*). Il est composé de deux parties, la partie *Déclaration* et la partie *Prédicat*. La partie *Déclaration* contient la déclaration des types des ensembles, relations, fonctions ... ainsi que les noms de schémas inclus. L'ensemble des variables du schéma forme sa signature. La partie *Prédicat* contient tous les prédicats définissant des propriétés sur les ensembles déclarés ou les opérations permettant de consulter ou de modifier les ensembles définis précédemment : cette partie définit donc la propriété du schéma. Cette partie est facultative, le prédicat par défaut est alors le prédicat *vrai*. Les définitions des ensembles déclarés dans un schéma sont locales, c'est pourquoi un schéma définissant une opération sur un ensemble défini dans un autre schéma doit inclure le nom du schéma. La forme horizontale d'un schéma est celle-ci :

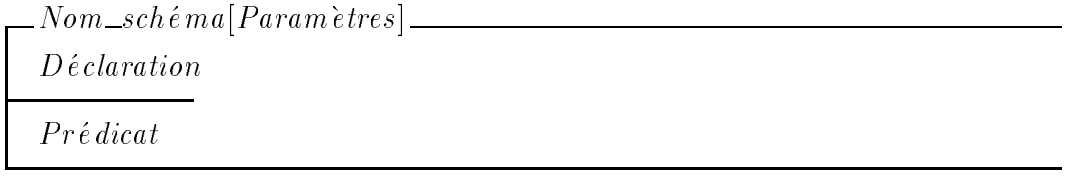
$Nom\_schéma \mathrel{\widehat=} [Déclaration \mid Prédicat]$

La définition axiomatique permet de définir un ensemble qui pourra être utilisé dans n'importe quel schéma sans faire appel à un autre schéma : cet ensemble est alors une variable globale. Elle est également composée d'une partie *Déclaration* et d'une partie *Prédicat*. Les propriétés définies dans la seconde partie d'une définition axiomatique ont toujours la valeur *vrai*, une définition axiomatique ne peut donc pas être utilisée pour définir les états successifs d'un système. La forme d'une définition axiomatique est la suivante :



Un prédicat peut apparaître tout seul. Il spécifie alors une contrainte sur des variables globales précédemment déclarées.

Les schémas génériques sont définis de façon similaire aux schémas ordinaires, mais avec en plus des paramètres génériques associés au nom du schéma :



Dans ce schéma, l'ensemble des types déclarés est étendu avec les paramètres. Lorsque

ce schéma sera utilisé dans la spécification, les paramètres devront être remplacés par des types déclarés.

Les constantes génériques ont une forme similaire aux schémas génériques mais sans le nom de schéma, et avec une double barre :

$$[Paramètres]$$
$$Déclaration$$
$$Prédicat$$

Comme pour le schéma générique, les paramètres doivent être remplacés par des types déclarés lorsqu'une définition générique est utilisée dans une spécification.

Une constante peut être déclarée par l'intermédiaire d'une définition axiomatique unique, sous la forme suivante :

$$Déclaration$$
$$Prédicat$$

Par exemple :

$$\pi : \mathbb{R}$$
$$\pi = 3,1415926535\ldots$$

A.4.2 Les opérateurs sur schémas

Les opérateurs sur schémas permettent de construire de nouveaux schémas à partir de schémas déjà définis dans la spécification. Deux schémas peuvent être combinés si leurs signatures (les variables) sont compatibles. Deux signatures sont compatibles si toute variable commune aux deux signatures a le même type dans chacune d'elles. La signature du schéma construit comprend alors l'ensemble des signatures des schémas combinés.

La propriété d'un schéma est une combinaison des propriétés des schémas combinés. Dans les descriptions suivantes des schémas, P_R, P_S et P_T sont respectivement les propriétés des schémas R, S et T.

$T \mathrel{\widehat{=}} \neg\, S$ — le schéma T est la négation de schéma S. La partie déclarative de S est la même que celle de T. P_T a pour valeur *vrai* si et seulement si P_S a pour valeur *faux*

$R \mathrel{\widehat=} S \wedge T$	le schéma R est la conjonction des schémas S et T. P_R a pour valeur *vrai* si et seulement si $P_S \wedge P_T$ a pour valeur *vrai*
$R \mathrel{\widehat=} S \vee T$	le schéma R est la disjonction des schémas S et T. P_R a pour valeur *vrai* si au moins une des propriétés de S ou de T est *vrai* : $P_R = P_S \vee P_T$
$R \mathrel{\widehat=} S \Rightarrow T$	le schéma R est égal à l'implication de schémas : S implique T. P_R a pour valeur *vrai* si et seulement si $P_S \Rightarrow P_T$ a pour valeur *vrai*
$R \mathrel{\widehat=} S \Leftrightarrow T$	le schéma R est égal à l'équivalence des schémas S et T. P_R a pour valeur *vrai* si et seulement si $P_S \Leftrightarrow P_T$ a pour valeur *vrai*

Les opérateurs précédents sont les opérateurs les plus utilisés pour combiner des schémas. Les opérateurs suivants ne sont ici que présentés, le lecteur se reportera à [Spivey94] pour plus d'explications.

$R \mathrel{\widehat=} \forall D \mid P \bullet S$	quantificateur universel de schéma
$R \mathrel{\widehat=} \exists D \mid P \bullet S$	quantificateur existentiel de schéma
$R \mathrel{\widehat=} S \setminus (x_1 \ldots x_n)$	masquage de schéma
$R \upharpoonright S$	projection de schéma
$R \mathrel{\widehat=} \mathrm{pre}\ S$	précondition de schéma
$R \mathrel{\widehat=} S \fatsemi R$	composition séquentielle de schémas
$R \mathrel{\widehat=} S \gg T$	tubage de schémas

A.4.3 Les systèmes séquentiels

Le langage Z est souvent utilisé pour spécifier un système séquentiel, c'est-à-dire un système comportant des opérations permettant de modifier un état. L'état du système est alors décrit par un ou plusieurs schémas. Les opérations sont elles aussi décrites par des schémas. Ces schémas ne décrivent pas comment se font ces opérations (structures algorithmiques) mais l'état atteint après cette opération. Ils décrivent également les conditions à partir desquelles peuvent se réaliser ces opérations. Par convention, la distinction entre les deux états d'un individu, d'un ensemble, etc se fait par l'utilisation de l'apostrophe : dans un schéma, tout composant sans apostrophe représente ce composant avant l'opération tandis que tout composant avec apostrophe représente ce composant dans l'état après opération.

L'opération décrivant l'initialisation est, par définition, une opération qui a un état après, mais n'a pas d'état avant. Le schéma décrivant cette opération inclut donc seulement comme schéma le schéma décrivant l'état après. Quant aux composants de ce schéma, ils sont pour, la même raison, tous décorés d'une apostrophe. Par exemple :

InitialisationFichier

$$Fichier'$$

$$contenu' = \emptyset$$

Les autres opérations nécessitent généralement d'inclure l'état du système avant et après l'opération. Certaines opérations transforment l'état du système, d'autres ne servent qu'à consulter cet état. Pour que la déclaration des schémas soit plus simple, il existe deux conventions correspondant à ces deux familles d'opérations : Δ et Ξ. Supposons que le schéma *Etat* représente l'état du système spécifié. Le schéma $\Delta Etat$ est alors défini comme étant la combinaison des schémas *Etat* et *Etat'*. Le schéma $\Xi Etat$ est aussi défini comme la combinaison des schémas *Etat* et *Etat'* avec en plus la condition que tous les composants de *Etat'* soient égaux à ceux correspondants de *Etat* (ce qui s'écrit $\theta Etat' = \theta Etat$).

A.4.4 Les schémas utilisés comme type

Un schéma peut être utilisé comme type dans une spécification Z. Reprenons l'exemple précédent de l'interface de programmation d'un système de gestion de fichier. Nous avons :

Fichier

$$contenu : clé \pfun donnée$$

Considérons maintenant un système de fichier qui gère 2 fichiers, décrits chacun par un schéma de type *Fichier*. Ce système peut être décrit, par exemple, par le schéma :

Système

$$fichier1 : Fichier$$
$$fichier2 : Fichier$$

$$\mathrm{dom}(fichier1.contenu) \cap \mathrm{dom}(fichier2.contenu) = \emptyset$$

Les variables *fichier*1 et *fichier*2 sont du type *Fichier*. Pour accéder à leurs constituants, il faut précéder l'identifiant de ces constituants par l'identifiant du schéma. Par exemple $fichier1.contenu$ est du type $clé \pfun donnée$, de même que $fichier2.contenu$.

Considérons maintenant un système de gestion de fichier dans lequel les fichiers sont gérés par leur nom. Dans ce cas nous pourions, par exemple, écrire :

$[Nom]$

Système

$$fichier : Nom \pfun Fichier$$

Les deux schémas *Système* correspondent à deux utilisations possibles d'un schéma comme type. Elles sont faciles d'utilisation et très pratiques. Dans [Woodcock et al.96], le lecteur trouvera un exemple détaillé de gestionnaire de fichier utilisant le schéma *File* comme type.

Annexe B

Autre méta-modèle possible du RdP généralisé

B.1 Les différences

Le méta-modèle que nous avons présenté dans le chapitre 4 se voulait le plus proche possible de la définition de Murata. Nous avons vu que des choix, qui peuvent être discutés, ont été faits. Ce nouveau méta-modèle est différent du méta-modèle précédent car il reprend l'ensemble des modifications possibles évoquées précédemment, c'est-à-dire :

- les ensembles P et T ne sont pas définis, d'où :
  - la règle spécifiant qu'il doit exister au moins une place ou une transition est transformée en : il doit exister au moins un arc,
  - le marquage ne peut concerner que des individus des ensembles $arcTP$ et $arcPT$ (ou plutôt du domaine du domaine de arcPT et de l'image du domaine de arcTP),
- le poids des arcs est directement intégré à la définition des arcs.

Ce méta-modèle est à comparer avec celui présenté dans la section 4.4, page 104, les différents cas de non respect de la précondition n'étant pas pris en compte pour ne pas surcharger le modèle. Les parties en caractères gras sont celles qui sont différentes de celles du méta-modèle précédent.

B.2 Méta-modèle complet

$[PLACE, TRANSITION]$

$USE ::= build \mid play$

PN

$\mathbf{arcTP} : \mathbf{TRANSITION} \times \mathbf{PLACE} \pfun \mathbb{N}_1$
$\mathbf{arcPT} : \mathbf{PLACE} \times \mathbf{TRANSITION} \pfun \mathbb{N}_1$
$M0 : PLACE \pfun \mathbb{N}$
$M : PLACE \pfun \mathbb{N}$
$enabled : \mathbb{F}\ TRANSITION$
$usePN : USE$

$\mathrm{dom}(\mathbf{M}) = \mathrm{ran}(\mathrm{dom}(\mathbf{arcTP})) \cup \mathrm{dom}(\mathrm{dom}(\mathbf{arcPT}))$
$\mathrm{dom}(\mathbf{M0}) = \mathrm{ran}(\mathrm{dom}(\mathbf{arcTP})) \cup \mathrm{dom}(\mathrm{dom}(\mathbf{arcPT}))$

PNinitial

PN'

$arcTP' = \{\}$
$arcPT' = \{\}$
$M0' = \{\}$
$M' = \{\}$
$enabled' = \{\}$
$usePN' = build$

AddarcTP

ΔPN
$OutputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$

$usePN = build$
$\mathbf{arcTP'} = \mathbf{arcTP} \oplus \{(\mathbf{Transition}? \mapsto \mathbf{OutputPlace}?) \mapsto \mathbf{Weight}?\}$
$arcPT' = arcPT$
$\mathbf{M0'} = \{\mathbf{OutputPlace}? \mapsto \mathbf{0}\} \oplus \mathbf{M0}$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddarcPT

ΔPN
$InputPlace? : PLACE$
$Transition? : TRANSITION$
$Weight? : \mathbb{N}_1$

$usePN = build$
$arcTP' = arcTP$
$\mathbf{arcPT'} = \mathbf{arcPT} \oplus \{(\mathbf{InputPlace}? \mapsto \mathbf{Transition}?) \mapsto \mathbf{Weight}?\}$
$M0' = \{InputPlace? \mapsto 0\} \oplus M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

AddM0

ΔPN
$Place? : PLACE$
$NewMarquage? : \mathbb{N}$

$usePN = build$
$\mathbf{Place}? \in \mathrm{ran}(\mathrm{dom}(\mathbf{arcTP})) \cup \mathrm{dom}(\mathrm{dom}(\mathbf{arcPT}))$
$arcTP' = arcTP$
$arcPT' = arcPT$
$M0' = M0 \oplus \{Place? \mapsto NewMarquage?\}$
$M' = M0'$
$enabled' = \{\}$
$usePN' = usePN$

StartPlay

ΔPN

$usePN = build$
$\#arcTP + \#arcPT \geq 1$
$arcTP' = arcTP$
$arcPT' = arcPT$
$M0' = M0$
$M' = M0$
$\mathbf{enabled'} = \{\mathbf{t} : \mathbf{TRANSITION} \mid (\forall \mathbf{p} : \mathbf{PLACE} \bullet$
$\quad (\mathbf{p}, \mathbf{t}) \in \mathrm{dom}(\mathbf{arcPT'}) \land \mathbf{arcPT'}(\mathbf{p}, \mathbf{t}) < \mathbf{M'}(\mathbf{p})) \lor \mathbf{t} \notin \mathrm{ran}(\mathrm{dom}(\mathbf{arcPT'}) \bullet \mathbf{t}\}$
$usePN' = play$

Play

ΔPN
$Fired? : TRANSITION$

$usePN = play$
$Fired? \in enabled$
$arcTP' = arcTP$
$arcPT' = arcPT$
$M0' = M0$
$\forall \mathbf{p} : \mathbf{PLACE} \bullet$
$\quad (\mathbf{p} \mapsto \mathbf{Fired}? \in \mathrm{dom}(\mathbf{arcPT}) \land \mathbf{Fired}? \mapsto \mathbf{p} \in \mathrm{dom}(\mathbf{arcTP})$
$\quad \land \mathbf{M'}(\mathbf{p}) = \mathbf{M}(\mathbf{p}) - \mathbf{arcPT}(\mathbf{p} \mapsto \mathbf{Fired}?) + \mathbf{arcTP}(\mathbf{Fired}? \mapsto \mathbf{p}))$
$\quad\quad \lor$
$\quad (\mathbf{p} \mapsto \mathbf{Fired}? \in \mathrm{dom}(\mathbf{arcPT}) \land \mathbf{Fired}? \mapsto \mathbf{p} \notin \mathrm{dom}(\mathbf{arcTP})$
$\quad \land \mathbf{M'}(\mathbf{p}) = \mathbf{M}(\mathbf{p}) - \mathbf{arcPT}(\mathbf{p} \mapsto \mathbf{Fired}?))$
$\quad\quad \lor$
$\quad (\mathbf{p} \mapsto \mathbf{Fired}? \notin \mathrm{dom}(\mathbf{arcPT}) \land \mathbf{Fired}? \mapsto \mathbf{p} \in \mathrm{dom}(\mathbf{arcTP})$
$\quad \land \mathbf{M'}(\mathbf{p}) = \mathbf{M}(\mathbf{p}) + \mathbf{arcTP}(\mathbf{Fired}? \mapsto \mathbf{p}))$
$\quad\quad \lor$
$\quad (\mathbf{p} \mapsto \mathbf{Fired}? \in \mathrm{dom}(\mathbf{arcPT}) \land \mathbf{Fired}? \mapsto \mathbf{p} \notin \mathrm{dom}(\mathbf{arcTP})$
$\quad \land \mathbf{M'}(\mathbf{p}) = \mathbf{M}(\mathbf{p}))$
$\mathbf{enabled'} = \{\mathbf{t} : \mathbf{TRANSITION} \mid (\forall \mathbf{p} : \mathbf{PLACE} \bullet$
$\quad (\mathbf{p}, \mathbf{t}) \in \mathrm{dom}(\mathbf{arcPT'}) \land \mathbf{arcPT'}(\mathbf{p}, \mathbf{t}) < \mathbf{M'}(\mathbf{p})) \lor \mathbf{t} \notin \mathrm{ran}(\mathrm{dom}(\mathbf{arcPT'}) \bullet \mathbf{t}\}$
$usePN' = usePN$

StopPlay

ΔPN

$usePN = play$
$arcTP' = arcTP$
$arcPT' = arcPT$
$M0' = M0$
$M' = M0'$
$enabled' = \{\}$
$usePN' = build$

Bibliographie

[Abrial96] Abrial (Jean-Raymond). – *The B-Book, assigning programs to meanings.* – Cambridge University Press, 1996.

[Adepa81] ADEPA. – *GEMMA : Guide d'Etude des Modes de Marches et d'Arrêts.* – 1981.

[ADPM92] *Automatisation des procédés mixtes continus et séquentiels.* – AFCET - SEE, 29-30 janvier 1992.

[Afnor75] *Symboles et conventions pour les organigrammes des données et les organigrammes de programmation.* – Norme n° Z 67–010, Association Française de NORmalisation - Union Technique de l'Electricité, 1975.

[Afnor82] *Diagramme fonctionnel GRAFCET pour la description des systèmes logiques de commande.* – Norme n° C 03–190, Association Française de NORmalisation - Union Technique de l'Electricité, 1982.

[Afnor93] *Diagramme fonctionnel GRAFCET.* – Norme n° C 03–191, Association Française de NORmalisation - Union Technique de l'Electricité, 1993.

[Afnor96] *Représentation des systèmes de contrôle et de commande des systèmes automatisés.* – Norme n° Z 68–901, Association Française de NORmalisation, 1996.

[Andreu et al.96] Andreu (David), Pascal (Jean-Claude) et Valette (Robert). – Events as a key of a batch process control system. *IMACS Multiconference CESA'96 Computational Engineering in Systems Applications.* pp. 297–302. – IEEE-SMC, 9-12 juillet 1996.

[Andreu et al.98] Andreu (David), Pascal (Jean-Claude) et Valette (Robert). – Supervision des systèmes de production discontinus. *Journal Européen des Systèmes Automatisés*, vol. 32, n° 3, 1998, pp. 365–386.

[Andreu96] Andreu (David). – *Commande et supervision des procédés discontinus : une approche hybride.* – Thèse de PhD, Université Paul Sabatier de Toulouse, 1996.

[Apave92] APAVE (Télémécanique) (édité par). – *La sûreté des machines et installations automatisées.* – 1992.

[Aristide90] Aristide (M.). – Modéliser un logiciel à l'aide de vdm : une expérience. *Technique et science informatiques*, vol. 9, n° 4, 1990, pp. 313–330.

[Barden et al.94] Barden (Rosalind), Stepney (Susan) et Cooper (David). – *Z in practice.* – Prentice Hall, 1994.

[Blanchard79] Blanchard (M.). – *Comprendre, maîtriser et appliquer LE GRAFCET.* – Cépaduès - éditions, 1979.

[Bodart et al.96] Bodart (Alain), Boissier (Raymond) et Dima (Bruno). – Enseigner l'informatique industrielle à l'aide d'un outil de développement orienté grafcet. *Modélisation des systèmes réactifs* [MSR96], pp. 381–388.

[Bon bierel94] Bon-Bierel (Evelyne). – *Méta-modèles du Grafcet.* – Rapport de recherche, Université de Nancy I - ENS de Cachan, 1994.

[Bon bierel98] Bon-Bierel (Evelyne). – *Contribution à l'intégration des modèles de systèmes de production manufacturière par méta-modélisation.* – Thèse de PhD, Université de Nancy I, 19 novembre 1998.

[Bouazza95] Bouazza (Med). – *Le langage EXPRESS.* – HERMES, 1995.

[Bouteille et al.95] Bouteille (Noël), Brard (Paul), Colombari (Gérard), Cotaina (Norberto) et Richet (Daniel). – *Le grafcet.* – Cépaduès-Editions, 1995.

[Bowen et al.93] Bowen (Jonathan) et Stavridou (Victoria). – Systèmes à sûreté de fonctionnement critique : méthodes formelles et normes. *Génie logiciel et systèmes experts*, n° 30, mars 1993.

[Bowen96] Bowen (Jonathan). – *Formal specification and documentation using Z: a case study approach.* – International Thomson Computer Press, 1996.

[Calvez90] Calvez (J. P.). – *Spécification et conception des systèmes - Une méthodologie : MCSE.* – Masson, 1990.

[Cei88] *Etablissement des diagrammes fonctionnels pour systèmes de commande.* – Norme n° CEI/IEC 848, Commission Electrotechnique Internationale, 1988.

[Champagnat98] Champagnat (Ronan). – *Supervision des systèmes discontinus : définition d'un modèle hybride et pilotage en temps réel.* – Thèse de PhD, Université Paul Sabatier de Toulouse, 1998.

[Chauvet96] Chauvet (J.-Y.). – Une étude de cas en B : les feux tricolores. *1st conference on the B method*, éd. par Habrias (Henri). Institut de recherches en informatique de Nantes, pp. 329–351. – 1996.

[Chen76] Chen (P.P.). – The entity-relationships model : toward an unified view of data. *ACM trans.on data base systems*, vol. 1, n° 1, 1976, pp. 9–36.

[Chlique92] Chlique (Pierre). – Spécification et conception des systèmes automatisés : les outils du génie automatique. *Revue générale de l'électricité*, n° 4, 1992, pp. 29–35.

[Coad et al.92] Coad (P.) et Yourdon (E.). – *Analyse orientée objets.* – Masson, 1992.

[Consortium89] Consortium (AMICE) (édité par). – *Open system architecture for CIM, Research reports Esprit project 688.* – AMICE Consortium - Springer verlag, 1989.

[Couffin et al.96] Couffin (Florent), Duméry (Jean-Jacques), Faure (Jean-Marc) et Frachet (Jean-Paul). – La méta-modélisation pour l'intégration des outils-méthodes : application au gemma et au grafcet. *Modélisation des systèmes réactifs* [MSR96], pp. 381–388.

[Couffin et al.98] Couffin (Florent), Lampérière (Sandrine) et Faure (Jean-Marc). – Contribution to the Grafcet formalisation : a static meta-model proposition. *Journal européen des systèmes automatisés*, 1998.

[Couffin97] Couffin (Florent). – *Modèles de données de référence et processus de spécialisation pour l'intégration des activités de conception en Génie Automatique.* – Thèse de PhD, Ecole Normale Supérieure de Cachan, 1997.

[Courtier et al.93] Courtier (Rémy), Hérin-Aime (Danièle) et Galéra (Richard). – Une démarche et un modèle de conception à base d'objets et de réseaux sémantiques. *Technique et science informatiques*, vol. 12, n° 3, 1993, pp. 285–318.

[David et al.89] David (R.) et Alla (H.). – *Du Grafcet aux réseaux de Pétri.* – Hermès, 1989.

[Denis et al.92] Denis (Bruno), Lesage (Jean-Jacques), Roussel (Jean-Marc) et Timon (Guy). – *Cahier des charges au format Guide d'Analyse des Besoins et modélisation IDEF 0, MERISE, GRAFCET et SA-RT - Etude de cas : SEDEM-91.* – Rapport technique, EXERA, 1992. LURPA - Ecole Normale Supérieure de Cachan.

[Denis et al.93] Denis (Bruno), Lesage (Jean-Jacques) et Timon (Guy). – Toward a theory of integrated modelling. *Science et technique de la conception*, vol. 2, n° 1, 1993, pp. 87–96.

[Denis94] Denis (Bruno). – *Assistance à la conception et à l'évaluation de l'architecture de conduite des systèmes de production complexes.* – Cachan, France, Thèse de PhD, Ecole Normale Supérieure de Cachan, février 1994.

[Diller94] Diller (Antony). – *Z: an introduction to formal methods.* – John Wikey and Sons, 1994, second édition.

[Gallimard92] Gallimard (édité par). – *Trésors de la langue française. Dictionnaire de la langue du XIXe et du XXe siècle.* – Centre National de Recherche Scientifique, 1992.

[Galloway et al.94] Galloway (A. J.) et O'Brian (S. J.). – *A formal abstract syntax for Ward-Mellor SA/RT Essential Models.* – Rapport technique, Université de Teesside, 1994. ftp://ftp.tees.ac.uk/pub/a.galloway/tech.report/tees-scm-4-94.ps.Z.

[Gaudel et al.96] Gaudel (Marie-Claude), Marre (Bruno), Schlienger (Françoise) et Bernot (Gilles). – *Précis de génie logiciel.* – Masson, 1996.

[Gee95] Gee (David M.). – Formal specification of visual languages. – 1995. http://lion.unn.ac.uk/ davidg/papers/fspec.ps.Z.

[Gerval et al.92] Gerval (Jean-Pierre), Guyot (Jean-Edouart), Lhoste (Pascal), Morel (Gérard) et Roesch (Michel). – MIAM : Modélisation Intelligente Appliquée à la Maintenance. In ADPM [ADPM92], pp. 161–172.

[Guegen et al.92] Guégen (Hervé), Chlique (Pierre) et Maurin (Serge). – Utilisation des stratecharts pour la spécification de la commande des systèmes dis-continus. In ADPM [ADPM92], pp. 161–172.

[Guegen et al.93] Guégen (Hervé) et Chlique (Pierre). – Users'requirements specification for control : an object-oriented approach. *7th european computer conference.* – Evry, France, 24–27 mai 1993.

[Habrias88] Habrias (Henri). – *Le modèle relationnel binaire.* – Eyrolles, 1988.

[Habrias93] Habrias (Henri). – *Introduction à la spécification.* – Masson, 1993, *Méthodologies du logiciel.*

[Habrias95] Habrias (Henri) (édité par). – *Z twenty years - what is its future ?* Institut de recherches en informatique de Nantes. – 1995.

[Harel87] Harel (David). – Statecharts : A visual formalism for complex systems. *Science of Computing Programming*, vol. 8, 1987, pp. 231–274.

[Iso93] *STEP 11 : The EXPRESS language reference manual.* – Norme n° ISO-10303-11, International Organization for Standardisation, 1993.

[iT89] igl Technologies (édité par). – *SADT : un langage pour communiquer.* – Eyrolles, 1989.

[Jia95a] Jia (Xiaoping). – *A Tutorial of ZANS - A Z animation system.* – Rapport technique, Chicago, Illinois, USA, DePaul University, 1995. ftp://ise.cs.depaul.edu/pub/.

[Jia95b] Jia (Xiaoping). – *ZTC : a type checker for Z notation. User's Guide.* – Rapport technique, Chicago, Illinois, USA, DePaul University, 1995. ftp://ise.cs.depaul.edu/pub/.

[Jones90] Jones (C. B.). – *Systematic software development using VDM.* – Prentice Hall, 1990, seconde édition.

[Kiefer et al.95] Kiefer (F.), Le Guirriec (C.), Michez (A.) et Gazzo (J. M.). – Les enjeux de la conception des Systèmes Intégrés de Production. *Congrès de Génie Industriel de Montréal.* – Montréal, 1995.

[Kiefer96] Kiefer (Francois). – *Contribution à l'ingénierie intégrée des systèmes de production : formalisation des mécanismes d'intégration entre modèles et applications sur site industriel.* – Cachan, France, Thèse de PhD, Ecole Normale Supérieure de Cachan, Janvier 1996.

[Lemoigne90] Le Moigne (Jean-Louis). – *La modélisation des systèmes complexes.* – Dunod, 1990, *Afcet Systèmes.*

[Lesage et al.92] Lesage (Jean-Jacques), Denis (Bruno) et Timon (Guy). – Une approche formelle de la modélisation intégrée. *La conception an 2000 et au-delà : outils et technologie.* – Strasbourg, France, 24–27 novembre 1992.

[Lesage et al.93] Lesage (Jean-Jacques), Piétrac (Laurent) et Timon (Guy). – *Modélisation fonctionnelle des parties mécaniques des tranches nucléaires : analyse bibliographique, étude de cas RCV.* – Rapport technique, LURPA - Ecole Normale Supérieure de Cachan, 1993.

[Lesage94] Lesage (Jean-Jacques). – *Contribution à la formalisation des modèles et méthodes de conception des systèmes de production.* – LURPA - Ecole Normale Supérieure de Cachan, Thèse, Université de Nancy I, 1994.

[Lher et al.96] L'her (Dominique), Le Parc (Philippe) et Marcé (Lionel). – Modélisation et vérification du grafcet. *Modélisation des systèmes réactifs* [MSR96], pp. 381–388.

[Lhoste et al.93] Lhoste (Pascal), Panetto (Hervé) et Roesch (Michel). – GRAFCET : from syntax to semantics. *APII*, n° 1, 1993, pp. 127–141.

[Lhoste et al.97] Lhoste (Pascal), Faure (Jean-Marc), Lesage (Jean-Jacques) et Zaytoon (Janan). – Comportement temporel du grafcet. *Journal européen des systèmes automatisés*, vol. 31, n° 4, 1997, pp. 695–711.

[Lhoste94] Lhoste (Pascal). – *Contribution au génie automatique : concepts, modèles, méthodes et outils.* – Thèse, Université de Nancy I, 1994.

[Lightfoot94] Lightfoot (David). – *La spécification formelle avec Z.* – Teknea, 1994. traduit de Formal Specification Using Z par Henri Habrias et Pierre-Marie Delpech.

[Lissandre90] Lissandre (Michel). – *Maîtriser SADT.* – Armand Collin, 1990.

[LP94] Le Parc (Philippe). – *Apports de la méthodologie synchrone pour la définition et l'utilisation du langage grafcet.* – Thèse de PhD, Université de Rennes 1, 1994.

[Meisels et al.97] Meisels (Irwin) et Saaltink (Mark). – *The Z/EVES reference manual.* – Rapport technique n° TR-97-5493-03d, Ottawa, Ontario, Canada, ORA Canada, 1997. http://www.ora.on.ca/.

[Monin96] Monin (Jean-François). – *Comprendre les méthodes formelles : panorama et outils logiques.* – Masson, 1996. Collection technique et scientifique des télécommunications.

[Morel92] Morel (Gérard). – *Contribution à l'automatisation et à l'ingénierie des Systèmes Intégrés de Production.* – Thèse, Université de Nancy I, 1992.

[MSR96] *Modélisation des systèmes réactifs.* – Brest, AFCET, 28–29 mars 1996.

[Murata89] Murata (Tadao). – Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, vol. 77, n° 4, 1989.

[Niclet et al.96] Niclet (Marc) et Zaytoon (Janan). – Outil d'aide à l'élaboration de modèles simulables. *Forum des doctorants - Automatique, Génie Informatique, Image.* E3i, pp. 199–202. – 6–7 juin 1996.

[Pierra et al.95] Pierra (Guy) et Ait-Ameur (Yamine). – Spécification de modèles de données orientés objet dans le domaine technique : le langage EXPRESS. *Revue d'automatique et de productique appliquées*, vol. 8, n° 2–3, 1995.

[Pietrac et al.96] Piétrac (Laurent), Denis (Bruno) et Jean-Jacques (Lesage). – Formalization of the design of control systems. *Sixth International Symposium on Robotics and Manufacturing (ISRAM'96), Second World Automation Congress (WAC'96).* – Montpellier, France, 27–30 Mai 1996.

[Proth et al.96] Proth (Jean-Marie) et Xie (Xiaolan). – *A tool for design and management of manufacturing systems.* – John Wiley, 1996.

[Rolland et al.90] Rolland (C.), Foucaut (O.) et Benci (G.). – *Conception des systèmes d'information. La méthode REMORA.* – Eyrolles, 1990.

[Roussel94] Roussel (Jean-Marc). – *Analyse de grafcets par génération logique de l'automate équivalent.* – Cachan, France, Thèse, Ecole Normale Supérieure de Cachan, 1994.

[Saaltink95] Saaltink (Mark). – *The Z/EVES system.* – Rapport technique, Ottawa, Ontario, Canada, ORA Canada, 1995. ftp://ftp.ora.on.ca/pub/doc/z-eves-draft.ps.Z.

[Saaltink97] Saaltink (Mark). – *The Z/EVES user's guide.* – Rapport technique n° TR-97-5493-06, Ottawa, Ontario, Canada, ORA Canada, 1997. ftp://ftp.ora.on.ca/pub/doc/z-eves-draft.ps.Z.

[Sartor95] Sartor (Marc). – Utilisation conjointe des langages statechart et grafcet pour la spécification des modes de marche. *Revue d'automatique et de productique appliquées*, vol. 8, n° 2–3, 1995, pp. 351–356.

[Spivey94] Spivey (J. M.). – *La notation Z.* – Masson and Prentice Hall, 1994. traduit de l'anglais par M. Lemoine.

[Stoddart et al.95] Stoddart (Bill), Fencott (Clive) et Dunne (Steve). – Modelling hybrid systems in z. In Habrias [Habrias95], pp. 11–25.

[Tardieu et al.83] Tardieu (Hubert), Rochfeld (Arnold) et Colletti (René). – *La méthode Merise. Principes et outils.* – Les éditions d'organisation, 1983.

[Valentine95] Valentine (Samuel H.). – An algebraic introduction of real numbers into Z. In Habrias [Habrias95].

[Vernadat93] Vernadat (François). – CIMOSA : enterprise modelling and enterprise integration using a process-based approach. *IFIP*, pp. 65–84. – 1993.

[Wilson86] Wilson (B.). – *Systems : Concepts, methodologies and applications.* – John WILEY & SONS N. Y., 1986.

[Woodcock et al.96] Woodcock (Jim) et Davies (Jim). – *Using Z : specification, refinement and proof.* – Prentice-Hall, 1996, *International series in computer science.*

[Zanettin94] Zanettin (Marc). – *Contribution à une démarche de conception des systèmes de production.* – Thèse de PhD, Université Bordeaux I, 1994.

[Zaytoon et al.93] Zaytoon (Janan), Niel (E.), Mile (A.) et Jutar (A.). – A temporal SADT for automated manufacturing systems. *International conference on Industrial Engineering and Production Management*, pp. 154–164. – Mons (Belgique), 2–4 juin 1993.

[Zaytoon et al.97] Zaytoon (Janan), Lesage (Jean-Jacques), Marcé (Lionel), Jean-Marc (Faure) et Lhoste (Pascal). – Vérification et validation du grafcet. *Journal Européen des Systèmes Automatisés*, vol. 31, n° 4, 1997, pp. 713–740.

[Zaytoon93] Zaytoon (Janan). – *Extension de l'analyse fonctionnelle à l'étude de la sécurité opérationnelle des systèmes automatisés de production.* – Thèse de PhD, INSA de Lyon, 1993.

Résumé :

Les travaux présentés dans ce mémoire portent sur l'étude des langages et méthodes de conception des Systèmes Automatisés de Conception (SAP). Notre objectif est l'amélioration de la rigueur de la définition de ces langages et méthodes. Le moyen retenu est l'utilisation d'un langage formel, le langage Z, pour les méta-modéliser.

Dans une premier temps, nous présentons les travaux existants sur l'étude de l'activité de modélisation, afin de montrer l'intérêt de la méta-modélisation vis-à-vis de notre objectif.

Dans un deuxième temps, nous caractérisons les différents aspects que doit couvrir un méta-modèle pour représenter avec rigueur un langage ou une méthode.

Dans un troisième temps, nous présentons de quelle façon le langage Z permet de couvrir l'ensemble de ces besoins. Nous validons alors notre approche sur deux exemples. Le premier exemple est un langage de conception des systèmes à événements discrets : les réseaux de Petri généralisés. Le deuxième exemple est une méthode de conception de la commande des systèmes hybrides intégrant deux langages : les réseaux de Petri temporels à événements et les équations différentielles.

Mots clés :

SAP, langages, méthodes, méta-modèles, langage formel Z.

Abstract:

Works presented in this thesis focus on the languages and the methods used for the design of Automated Manufacturing Systems (AMS). The main objective is to improve the definition exactness of these languages and methods, and the selected approach consists in "meta-model" them using a formal language : the Z language.

In a first time, we present existent works on the modeling activity, in order to show the interest of the meta-modeling to answer our objective.

In a second time, we consider the different requirements that the meta-model has to ensure to correctly to represent a language or a method.

Then, we present in which way the Z formal language meets all these requirements.

The efficiency of the approach is validated through two examples. The first one consists in the meta-modeling of a language used for the design of the discreet event systems: the generalized Petri nets. For the second example, the approach is applied to a method used for the design of hybrid system controls which uses two languages: events temporal Petri nets and differential equations.

Key words:

AMS, languages, methods, meta-models, Z formal language.